Integrationsmöglichkeiten einer Swing-Anwendung in Eclipse am Beispiel des abaXX Process Modelers

Diplomarbeit im Studiengang Medieninformatik der
Fachhochschule Stuttgart – Hochschule der Medien

Autor: **Martin Brenda**

Vorgelegt: **28.02.2006**

1. Prüfer: **Prof. Dr. Johannes Maucher**

2. Prüfer: **Dipl. Math. Harald Gliebe**

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst habe und dass sämtliche Quellen im Text oder im Anhang nachgewiesen sind.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Martin Brenda
Stuttgart, 28. Februar 2006

Inhaltsverzeichnis

Danksagung

Danken möchte ich meinen Eltern, Marian und Irena, sowie meinem Bruder Peter, für die Unterstützung während des gesamten Studiums. Ohne ihren Rückhalt hätte ich das Studium nicht geschafft!

Abstract

The Eclipse framework established itself as an integrated development environment and platform for tool integration over the past years. More and more known companies like BEA, IBM and Nokia are integrating their applications in this framework. Eclipse provides for this a modular system, which can be extended easily.

This diploma thesis deals with the integration of applications in the Eclipse platform. Main attention lies in the integration of existing Swing applications into this framework. The basis therefor builds the Process Modeler, a Swing applcation from abaXX Technology AG for modelling business processes. Starting from the theoretical groundwork and the motivation for an integration, the Process Modeler and the Eclipse framework will be analyzed, a so some other used technologies. Goal is the study of the integration possobilities in the Eclipse framework and the setting up of an concrete integration scenario for the Process Modeler, as well as the creation of a prototype.

Kurzfassung

Das Eclipse-Framework hat sich in den letzten Jahren als Entwicklungsumgebung und Plattform für Tool-Integration etabliert. Immer mehr bekannte Firmen wie BEA, IBM und Nokia integrieren ihre Anwendungen in dieses Framework. Eclipse bietet hierfür ein modulares System an, das auf einfache Weise erweitert werden kann.

Die vorliegende Diplomarbeit beschäftigt sich mit der Integration von Anwendungen in die Eclipse-Plattform. Hauptsächliches Augenmerk liegt dabei auf der Integration vorhandener Swing-Applikationen in das Framework. Grundlage hierfür bildet der Process Modeler, eine Swing-Applikation von der abaXX Technology AG zum Modellieren von Geschäftsprozessen. Aufbauend auf den theoretischen Grundlagen und der Motivation für eine Integration werden zuerst der Process Modeler und das Eclipse Framework sowie einige weitere verwendete Technologien analysiert. Ziel ist die Untersuchung von Integrationsmöglichkeiten in das Eclipse Framework und das Aufstellen eines konkreten Integrationsszenarios für den Process Modeler, sowie die Erstellung eines ersten Prototypen.

Einleitung

„Wir produzieren heute Informationen in Massen, wie früher Autos."
- John Naisbitt -

Ein Problem mit dem sich Software-Entwickler seit einigen Jahren auseinander setzen müssen ist die beträchtliche Anzahl verschiedener Anwendungen für die immer weiter steigende Anzahl an Technologien. Als Reaktion darauf wurde Eclipse ins Leben gerufen. Eclipse ist bekannt als integrierte Entwicklungsumgebung und Plattform für Tool-Integration. Im Rahmen der Diplomarbeit werden die Integrationsmöglichkeiten in diese Plattform untersucht. Vor allem die Integration vorhandener Swing-Anwendungen ist von besonderem Interesse.

Die Grundlage der Diplomarbeit bildet der Process Modeler, eine von der abaXX Technology AG entwickelte, interaktive Anwendung zum grafischen Modellieren von Prozessen. Bei dieser Anwendung handelt es sich um eine eigenständige Swing-Applikation. Im Rahmen der Diplomarbeit soll untersucht werden, auf welche Art und Weise der Process Modeler am besten und einfachsten in Eclipse integriert werden könnte. Die Wunschlösung ist eine vollständige Integration und Ablösung der Swing-Applikation. Die beiden Hauptziele der Diplomarbeit sind die Festsetzung eines Integrationsszenarios für die zukünftige Entwicklung und die Erstellung eines ersten Prototypen. Die abaXX Technology AG erhofft sich von einer Integration in Eclipse die Produktivität beim Arbeiten mit dem Process Modeler steigern und zusätzliche, in Eclipse bereits vorhandene, Features nutzen zu können.

Der Entwicklungsprozess von systematisch entwickelter Software ist komplex und umfasst eine Vielzahl von Teildisziplinen der Informatik und des Projektmanagements. Die Disziplinen sind während des ganzen Entwicklungsprozesses eng miteinander verzahnt. Es wurde versucht innerhalb der Diplomarbeit diesen Softwareentwicklungsprozess so weit wie

möglich nachzuempfinden (siehe Abbildung 1). Von der Planung und Analyse über den Entwurf bis hin zur Programmierung inklusive Tests wird das gesamte Spektrum der Softwareentwicklung abgedeckt. Einer der wichtigsten Punkte war das iterative Vorgehen, um die Ergebnisse immer wieder zu analysieren und daraus neue Erkenntnisse für den weiteren Ablauf zu definieren.

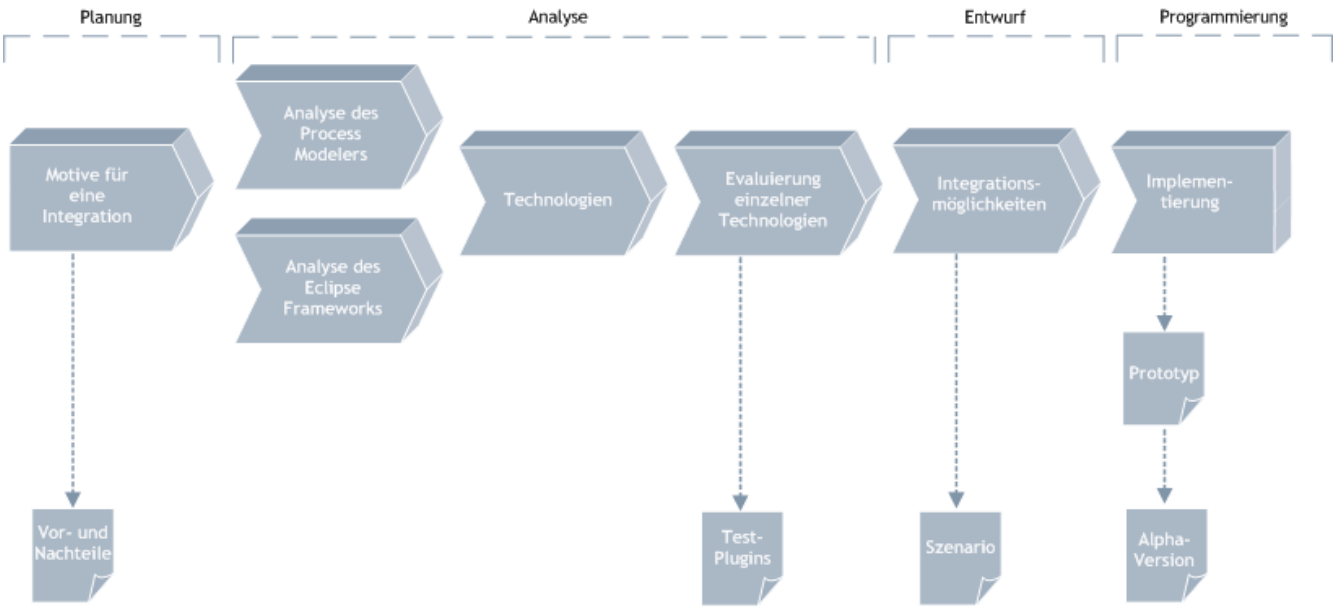

Abbildung 1: Vorgehen in der Diplomarbeit

Im ersten Kapitel werden die Motive für eine Integration betrachtet. Es sollen vor allem die Vor- und Nachteile einer Integration in Eclipse aufgezeigt und die erhofften Verbesserungen verifiziert werden. Im Anschluss daran folgt die Analyse der beiden Anwendungen welche den Grundstein der Diplomarbeit bilden. Zuerst wird der Process Modeler untersucht, seine Rolle im abaXX Workflow und vor allem sein Aufbau. Danach wird das Eclipse Framework beleuchtet, die Konzepte die hinter ihm stehen und die bestehenden Erweiterungsmöglichkeiten. Im Anschluss folgt die theoretische Betrachtung einiger grundlegender Technologien. Vor allem die grafischen Bibliotheken, AWT und SWT, sowie die grafischen Frameworks, ILOG JViews und das Graphical Editing Framework (GEF), sind von besonderem Interesse. Nach dieser theoretischen Analyse folgt die praktische Evaluierung einzelner Technologien. Das Ziel ist hierbei festzustellen, ob die Technologien die benötigten Anforderungen erfüllen. Darauf aufbauend werden die

Integrationsmöglichkeiten aufgezeigt und einige erfolgreiche Integrationen anderer Anwendungen angesehen. Hieraus ergibt sich ein konkretes Integrationsszenario für den Process Modeler. Den Abschluss der Diplomarbeit bildet die Implementierung des ersten und zweiten Integrationsschritts aus dem Integrationsszenario. Der erste Schritt wird sehr detailliert beschrieben, um die Vorgehensweise bei der Implementierung von Erweiterungen innerhalb des Eclipse Frameworks aufzuzeigen. Beim zweiten Schritt wird nur noch auf besonders interessante Inhalte eingegangen, da eine ausführliche Dokumentation dieses aufwändigen Integrationsschritts den Rahmen der Diplomarbeit sprengen würde.

abaXX Technology AG

Die vorliegende Diplomarbeit wurde bei der abaXX Technology AG in Stuttgart, Deutschland, durchgeführt. Die abaXX Technology AG kombiniert Kompetenz in Portallösungen mit Wissen im Business Process Management. Darauf aufbauend bietet sie komponentenbasierte und anpassbare Produkte im Bereich der Prozessportalsysteme auf J2EE-Basis an. Mit Hilfe dieser Produkte werden individuelle Kundenlösungen realisiert, von der Spezifikation über die Implementierung bis zum Rollout. Die abaXX Technology AG zählt in diesem Umfeld zu den führenden Anbietern in Europa mit starker Präsenz in Deutschland. Zu ihren mehr als 50 Kunden zählen unter anderem Cortal Consors und die DaimlerChrysler Bank.

Konventionen in der Diplomarbeit

Der Java-Code ist in `Schreibmaschinenschrift` gesetzt und befindet sich in einem Kasten. Konkrete Dateinamen sind ebenfalls in `Schreibmaschinenschrift` angegeben. Textverweise auf die Benutzeroberfläche sind bei Menübefehlen, Symbolleistenschaltflächen und Dialogfeldern durch KAPITÄLCHEN gekennzeichnet. Zum Beispiel bedeutet FILE | NEW | DOCUMENT, dass zuerst das Menü FILE geöffnet und dann NEW gefolgt von DOCUMENT gewählt wird. Optionen in Dialogfeldern und Eingaben des Benutzers sind in *Kursivschrift* angegeben.

Begleit-CD

Die zur Diplomarbeit gehörende CD enthält zusätzliches Material:

- Die Diplomarbeit im elektronischen Format
- Alle elektronischen Quellen inklusive gespeicherter Internetseiten
- Präsentationen die zum Thema der Diplomarbeit gehalten wurden
- Vollständiger Quellcode des ersten Integrationsschritts sowie zur Evaluierung der AWT-AWT-Brücke und des GEF
- Eine Eclipse-Installation mit dem ersten Integrationsschritts sowie den Plug-Ins zur Evaluierung einzelner Technologien
- Verschiedene Screenshots

1 Motive für eine Integration

„Optimism is an occupational hazard of programming: feedback is the treatment."
- Kent Beck -

Laut einer Leserbefragung von OBJEKTspektrum liegt die Bedeutung von Eclipse inzwischen höher als jene der service-orientierten Architektur (vgl. Spektrum, 2005, S. 5). Bei der Frage welche Themen für die Weiterbildung wichtig sind, stimmten fast 56 % für Eclipse. In 63,8 % der Fälle wird Eclipse bereits eingesetzt. Fast 11 % wollen Eclipse binnen eines Jahres nutzen.

Diese Leserberfragung zeigt, dass Eclipse sich in vielen Firmen als Entwicklungsumgebung etabliert hat. Mehr als das: zur Zeit gibt es einen regelrechten Hype um das Thema Eclipse. Neben dem Einsatz als Entwicklungsumgebung bietet die Eclipse Plattform auch die Möglichkeit sie als Fundament für eigene Anwendungen zu nutzen und diese dort zu integrieren. Im folgenden werden einige Vor- und Nachteile einer Integration in Eclipse betrachtet um die Motivation dieser Diplomarbeit näher zu bringen.

1.1 Vorteile

Software-Hersteller müssen die Akzeptanz ihrer Anwendungen maximieren und gleichzeitig die Entwicklungskosten reduzieren. Wie schon erwähnt erfreut sich die Eclipse Plattform zunehmender Beliebtheit und einer weiten Verbreitung. Es herrscht ein regelrechter Hype. Dies kann für andere Firmen ein Grund sein, auf Eclipse aufbauende Produkte zu favorisieren. Auch für Marketing-Zwecke lässt sich diese Situation ausnutzen. Viele bekannte Firmen wie BEA, IBM und seit neustem auch Nokia integrieren ihre Tools bereits in die Eclipse Plattform.

Hinter der Eclipse Plattform stehen viele große und bekannte Firmen. Die Plattform selbst ist Open Source und steht unter der Eclipse Public License (EPL). Diese Lizenz erlaubt eine effiziete kommerzielle Nutzung, da nicht der Zwang besteht neu entwickelte Plug-Ins auch unter diese Lizenz zu stellen, wie dies zum Beispiel bei der GPL der Fall ist. Eclipse ist kostenlos erhältlich, so dass für den Anwender keine Lizenzkosten entstehen. Der Quellcode kann eingesehen und ohne irgendwelche Begrenzungen angepasst werden.

Eclipse ist speziell für die Integration anderer Tools entwickelt worden. Es bildet das Fundament, auf dem andere Anwendungen aufbauen können. Dieses Fundament wird ständig weiterentwickelt und verbessert. Die Hersteller von Tools können sich somit auf die Entwicklung neuer Features konzentrieren, wodurch Zeit- und Kostenvorteile entstehen. Durch den modularen Aufbau von Eclipse, und letztlich auch des integrierten Tools, lassen sich einzelne Komponenten einfach austauschen. Auf diese Weise kann schneller und effizienter auf Technologieänderungen reagiert werden.

Eclipse unterstützt die Entwicklung im Verlauf des gesamten Lebenszyklus. Von der Analyse- und Designphase über die Implementierung bis hin zum Testen und dem Betrieb. Durch eine Integration erbt die eigene Anwendung die zusätzlichen Vorteile einer Basis-Entwicklungsumgebung. Viele Infrastruktur-Dienste wie Syntax-Highlighting oder Debugging werden bereits unterstützt und können mit der eigenen Anwendung verknüpft werden.

Einer der größten Vorteile im Fall des Process Modelers ist jedoch der direkte Zugriff auf die Modellierung und Implementierung aus einer Anwendung im Falle einer vollständigen Integration. Es muss nicht mehr zwischen mehreren Applikationen hin und her geschaltet werden, was auf Dauer sehr nervtötend sein kann. Es entfällt auch die ewige Sucherei nach der Implementierung zu einem bestimmten Modell-Element. Im Endeffekt resultiert daraus ein produktiverer Arbeitsablauf. Auch die homogene Benutzeroberfläche von Eclipse hat einen positiven Effekt auf den Arbeitsablauf. Denn bei ihrer Entwicklung spielte die Benutzerfreundlichkeit eine wichtige Rolle.

1.2 Nachteile

Die Integration einer Anwendung in Eclipse erfordert vor allem am Anfang finanzielle Investitionen. Im Fall des Process Modelers ist vor allem der Aufwand für das im späteren Verlauf geplante Austauschen von ILOG JViews durch das Graphical Editing Framework (GEF) nicht zu unterschätzen.

Die Eclipse Foundation bietet keinen Support für Eclipse. Die kann sowohl bei der Entwicklung der eigenen Anwendung problematisch sein als auch beim späteren Kundensupport. Die Firma muss dem Kunden auch Support für Eclipse selbst leisten. Ein weiterer negativer Faktor in dieser Hinsicht ist die weniger ausführliche Dokumentation im Vergleich zu kommerziellen Produkten. Zusätzlich bleibt die Frage der Haftung offen, da die Eclipse Foundation auch keine Haftung für das Eclipse Framework übernimmt. Dies ist allgemein üblich bei Open Source Software.

Eclipse ist noch nicht wirklich bereit für J2EE. Eine Anwendung die hauptsächlich auf diesen Bereich abzielt wird viele Dinge selbst implementieren müssen ohne auf vorhandene Frameworks zugreifen zu können.

Da Eclipse das Fundament der gesamten Anwendung bildet entsteht eine gewisse Abhängigkeit vom Eclipse Framework. Hierdurch kann die Eclipse Foundation die zukünftige Entwicklung der eigenen Anwendung und die Verwendung bestimmter Technologien in einem kleinen Rahmen beeinflussen. Ein Beispiel hierfür wäre zum Beispiel SWT. Würde die Swing-Bridge nicht mehr unterstützt müsste die eigene Anwendung vollständig in SWT umgesetzt werden.

Einer der größten, auf den ersten Blick oft nicht sichtbaren, Nachteile ist das kontinuierliche Anpassen der eigenen Anwendungen an die Eclipse Meilensteine, womit wiederum zusätzlicher Aufwand verbunden ist. Dieses Problem resultiert daraus, dass während der Weiterentwicklung von Eclipse immer wieder Änderungen an der API

vorgenommen werden. Diese Änderungen können nur schwer abgeschätzt werden. Es ist deshalb notwendig bei jedem Meilenstein die eigene Anwendung auf Funktionstüchtigkeit zu überprüfen.

1.3 Fazit

Es gibt eine Reihe von Gründen, die für eine Integration des Process Modelers in Eclipse sprechen. Vor allem die Zusammenführung zweier Anwendungen zu einer und die damit verbundenen Vorteile wiegen schwer. Aber auch die Fokussierung der zukünftigen Entwicklung auf neue Features und die damit einhergehenden Zeit- und Kostenvorteile sind ein wichtiger Faktor. Demgegenüber steht vor allem die ständige Anpassung der Anwendung an die Meilensteine von Eclipse. Insgesamt gesehen überwiegen aber die Vorteile den Nachteilen.

2 Analyse des Process Modelers

„The general precept of any product is that simple things should be easy, and hard things should be possible.“
- Alan Kay -

In diesem Abschnitt wird zunächst auf die abaXX.components und innerhalb dieser auf die process.component eingegangen, da diese die Grundlage des Process Modelers bilden und somit den technischen Rahmen für diese Diplomarbeit setzen. Anschließend wird der Process Modeler selbst genauer analysiert.

Die von der abaXX Technology AG angebotenen abaXX.components sind eine Sammlung von Diensten und Frameworks zur Erstellung von Enterprise Informationssystemen auf Basis von bewährten und standardisierten Technologien im Umfeld der Java 2 Enterprise Edition (J2EE). Die Base Edition zum Beispiel umfasst Komponenten für den Portalaufbau und die Benutzerverwaltung, die Erstellung des Portals mit vordefinierten Portlets und Funktionen für den Nachrichtenversand. Ferner wird unter anderem die process.component zur Integration von Geschäftsprozessen angeboten, die nicht Bestandteil der Base Edition ist.

Portale der kommenden Generation, so genannte Prozess-Portale, versuchen sich möglichst nahtlos in die Geschäftsprozesse eines Unternehmens zu integrieren (vgl. abaXX3, 2004). Mit der process.component wird ein deklarativer Ansatz zur Implementierung von Geschäftslogik eingeführt (vgl. abaXX1, 2005). Auf diese Weise wird eine freie und lose Kopplung von Arbeitsabläufen erreicht, anstelle fest definierter und unveränderbarer Geschäftsprozesse (vgl. abaXX2, 2004). Die process.component erlaubt die Modellierung und Ausführung von Prozessen, sowohl kurz laufender Microflows und Pageflows, als auch lang laufender Geschäftsprozesse mit mehreren Teilnehmern. Die Grundlage hierfür bildet eine Workflow-Technologie.

2.1 Der abaXX-Workflow

Ein Workflow ist ein Prozess (Arbeitsablauf), der aus einzelnen Aktivitäten aufgebaut ist, die sich auf Teile eines Geschäftsprozesses oder andere organisatorische Vorgänge beziehen. Eine Aktivität in Workflows bildet die kleinste Ausführungseinheit. Ihr sind typischerweise eine Tätigkeit, ausführende Ressourcen, zu benutzende Ressourcen und zeitliche Abhängigkeiten zugeordnet. Der Workflow beschreibt ausführlich die operative Ebene, idealerweise so exakt, dass die folgende Aktivität durch den Ausgang der jeweils vorangegangenen determiniert ist. Die einzelnen Aktivitäten stehen demnach in Abhängigkeit zueinander. Ein Workflow hat einen definierten Anfang, einen organisierten Ablauf und ein definiertes Ende. Allgemein sind Workflows oragnisationsweite arbeitsteilige Prozesse, in denen die anfallenden Tätigkeiten von Personen oder Software-Systemen koordiniert abgearbeitet werden.

2.1.1 Prozessmodellierung und -ausführung

Die process.component ermöglicht eine ganzheitliche Sicht auf die Geschäftsprozesse, von der Definition über die Ausführung bis zur Kontrolle (vgl. abaXX2, 2004). Die Prozessmodellierung und -ausführung bilden die grundlegenden Phasen bei der Einbindung von Geschäftsprozessen.

Während der Modellierung werden die Geschäftsprozesse mit Hilfe des Process Modelers als Folge von Aktivitäten grafisch definiert und erstellt (siehe Abbildung 2). Dem Benutzer stehen hierzu vorprogrammierte Arbeitsschritte (Aktivitäten) zur Verfügung aus denen er die benötigten auswählt und miteinander zu einem kompletten Prozessablauf verknüpft. Die verfügbaren Aktivitäten sind mit Hilfe von Java-Klassen implementiert. Einige oft verwendete werden bereits mit den abaXX.components mitgeliefert, ansonsten aber im Projekt kundenspezifisch programmiert. Hieraus ergibt sich ein Baukasten für die Workflows im Portal. Das Design der Prozesse erfolgt aus Benutzer-Sicht, nicht aus technischer Sicht. Die Modellierung wird meist von Fachleuten durchgeführt, die wenig

Interesse an der eigentlichen Implementierung haben. Die Ergebnisse werden als Prozess-Definitionen in einem XML-Format gespeichert und sind ohne weitere Programmierarbeit direkt vom Portal ausführbar (vgl. abaXX3, 2004). Vorteile dieser Vorgehensweise sind eine schnelle und flexible Reaktion auf neue bzw. veränderte Anforderungen, sowie eine Dokumentation der Portal-Prozesse.

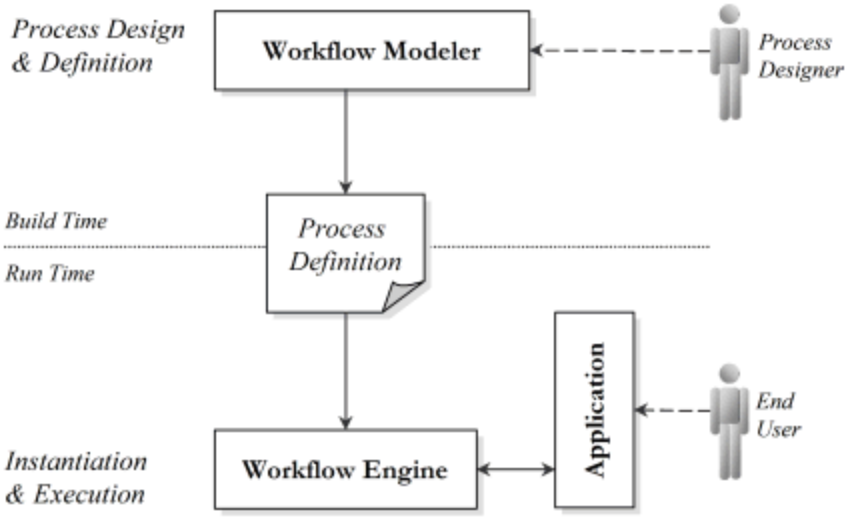

Abbildung 2: Prozessmodellierung und -ausführung (Quelle: abaXX Technology AG)

Zur Laufzeit werden die mit dem Process Modeler erstellten Prozess-Definitionen von der Workflow-Engine geladen und in ausführbare Prozess-Instanzen umgewandelt (siehe Abbildung 2). Während der Ausführung eines Prozesses ruft das System die deklarierten Aktivitäten auf. Die Implementierung einer Aktivität, konkret die Java-Klassen, lesen und schreiben Daten vom und in den Prozess-Kontext. Daraus entsteht ein Datenfluss der über die weitere Richtung des Kontrollflusses entscheidet (vgl. abaXX4, 2005). Die Workflow-Engine verarbeitet die Geschäftslogik und erzeugt die resultierenden Ergebnisse. Darüberhinaus kann sie Pages anzeigen und auf die Eingabe des Benutzers warten (Pageflow), sowie Prozesse in ihrem Ablauf unterbrechen und für den Eintritt von bestimmten Ereignissen registrieren (Event-Registry). Weitere Aufgaben der Workflow-Engine sind das Anstoßen von Sub-Prozessen und deren Verarbeitung. Um langlebige Prozesse abbilden zu können werden suspendierte Workflows automatisch persistent

gehalten. Durch diese Flexibilität sind spätere Änderungen an einem Geschäftsprozess ohne Modifikationen des Programmcodes möglich, wodurch geringere Betriebskosten entstehen. Auch die Einbindung von technisch weniger erfahrenen Mitarbeitern in den Arbeitsablauf ist ein großer Vorteil.

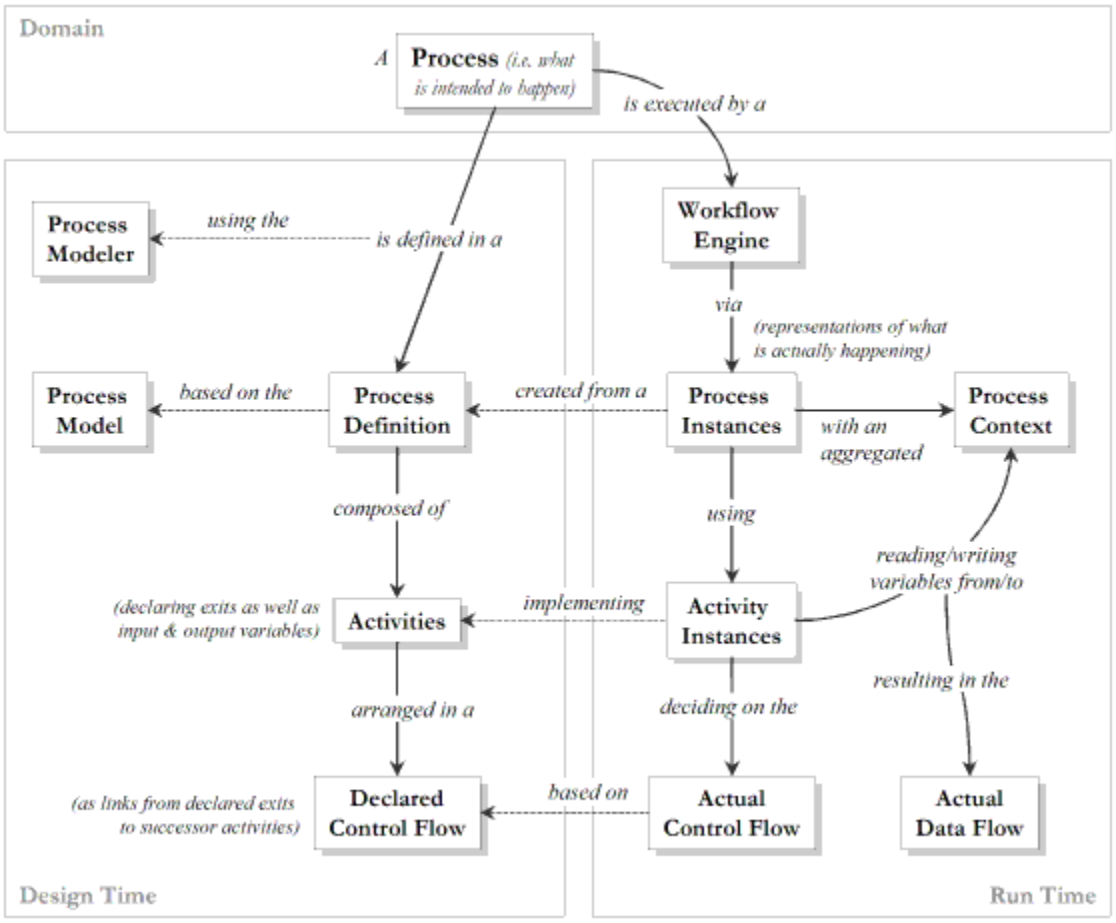

Abbildung 3: Konzept des abaXX-Workflows (Quelle: abaXX Technology AG)

Ein Prozess wird zur Laufzeit durch eine Prozess-Instanz repräsentiert. Erst mit Hilfe dieser Instanz ist es möglich den Prozess auszuführen (siehe Abbildung 3). Durch den Kontrollfluss wird die Reihenfolge der verschiedenen Aktivitäten im Prozess definiert. Die Übergabe der Informationen zwischen den einzelnen Aktivitäten wird durch den Datenfluss definiert. Je nach Ausführungsergebnis innerhalb einer Aktivität kann der Kontrollfluss unterschiedliche Wege einschlagen.

2.1.2 Prozesse & Aktivitäten

Im Vergleich zu Standards wie BPEL oder BPML deckt der abaXX Workflow nur die fundamentalen Konzepte der Prozessmodellierung ab. Alle komplizierteren Vorgänge wie Bedingungen oder Schleifen werden durch fertig gebaute Aktivitäten gekapselt. Das Meta-Modell und die fertigen Aktivitäten bilden das Modell der Prozessdefinition. Dieses Modell kann durch eigene Aktivitäten angepasst und erweitert werden um spezielle Anforderungen eines Projekts erfüllen zu können. Durch das einfache Prozessmodell kann die Workflow-Engine ein Ressourcen schonendes Laufzeitsystem einsetzen was in geringem Overhead resultiert. Dadurch ist es möglich sogar extrem kurz laufende Microflows auszuführen, die nur wenige Millisekunden laufen. Viele andere Workflow-Engines können nur lang laufende Prozesse ablaufen lassen, da sich ihr Einsatz durch den produzierten Overhead bei kurz laufenden Prozessen nicht kalkuliert. Ein Prozess besteht aus unterschiedlichen Aktivitäten die einem Kontrollfluss folgen und den Datenfluss beeinflussen (vgl. abaXX4, 2005). Prozesse lassen sich als Sub-Prozesse auch ineinander schachteln.

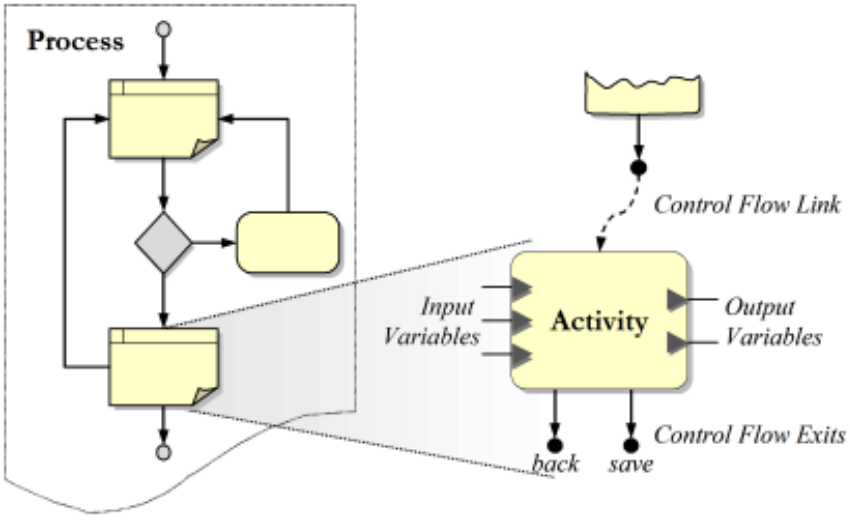

Abbildung 4: Prozess Meta Model (Quelle: abaXX Technology AG)

Aktivitäten sind essentielle Bestandteile von Prozessen. Sie kapseln komplexe Funktionalität und stellen nach Außen öffentliche Schnittstellen für Eingabe- und Ausgabeparameter, sowie Kontrollflussausgänge zur Verfügung (siehe Abbildung 4). Die

einzelnen Aktivitäten sind unabhängig voneinander, was die Wiederverwendbarkeit erhöhen soll. Ihre Implementierung besteht aus Java-Klassen. Zusammen mit den abaXX.components kommen eine Reihe von Aktivitäten, die oft verwendete Funktionen als Bausteine für Workflows zur Verfügung stellen (vgl. abaXX3, 2004).

2.2 Der Process Modeler

Der Process Modeler ist eine interaktive Anwendung zur grafischen Modellierung von Geschäftsprozessen (vgl. abaXX3, 2004). Mit ihm können neue Prozesse schnell und effizient erstellt und vorhandene editiert werden (vgl. abaXX5, 2004, S. 8). Aktivitäten lassen sich einfach entwickeln und Sub-Prozesse miteinander verknüpfen. Auch das Ausliefern (Deployment) der fertigen Prozesse wird vom Process Modeler unterstützt. Er kann mit und ohne abaXX.components installiert werden.

Der Process Modeler speichert die erstellten Prozesse als XML-Dateien ab. Neben den von der Workflow-Engine benötigten Werten für den Ablauf des Workflows werden auch die für die grafische Darstellung verantwortlichen Attribute in die XML-Datei gespeichert.

2.2.1 Aufbau

Der Process Modeler ist eine eigenständige Swing-Anwendung. Nach dem Start erscheint das Hauptfenster, das in einzelne Unterfenster aufgeteilt ist (siehe Abbildung 5). Im Hauptfenster gibt es drei Bereiche zwischen denen einige der Unterfenster hin- und hergeschoben und an anderer Stelle angedockt werden können. Die Aufteilung der Oberfläche und die Anordnung der einzelnen Elemente macht einen guten Eindruck. Die wichtigsten Elemente des Hauptfensters werden im folgenden kurz beschrieben (vgl. abaXX, S. 12f.).

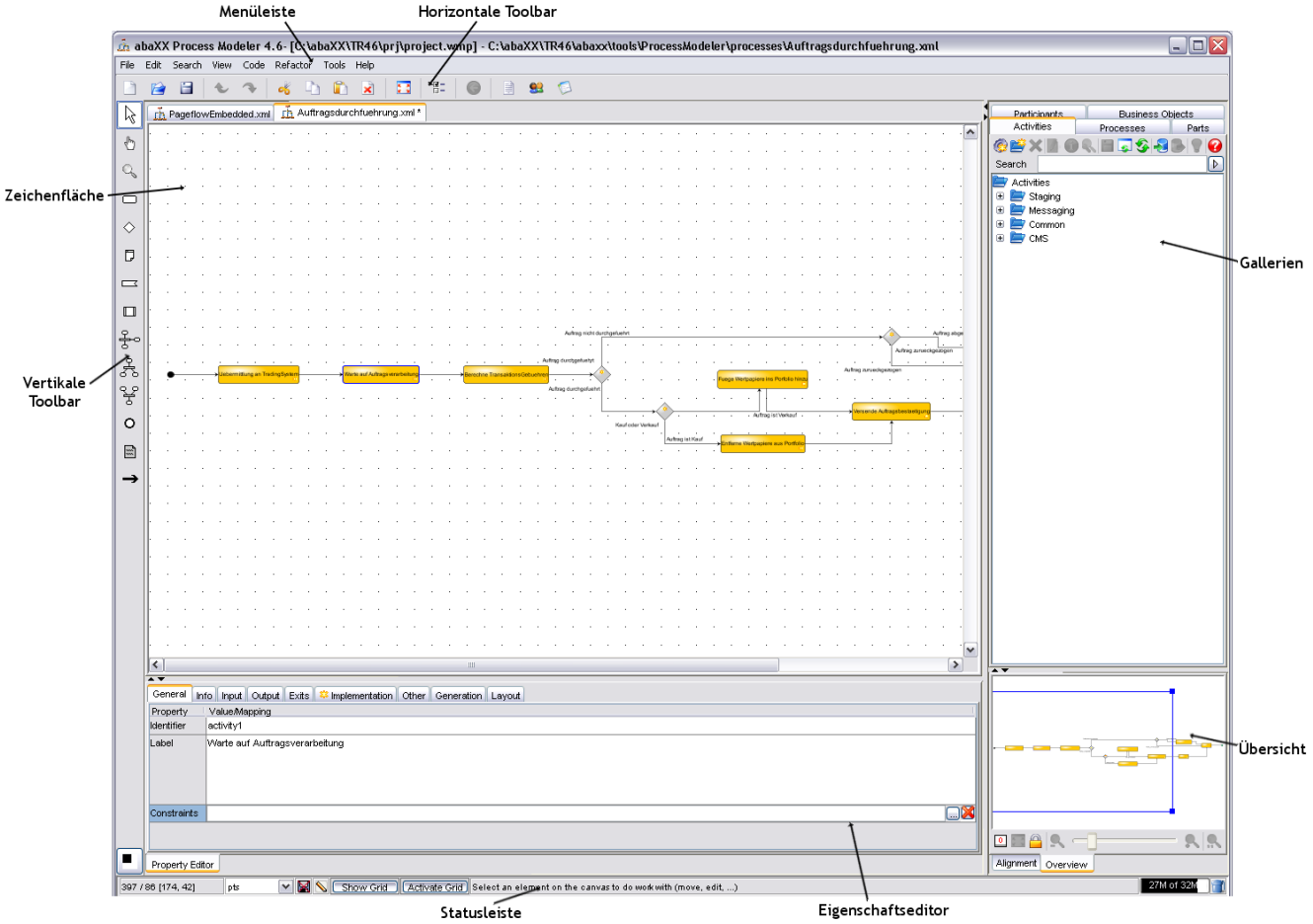

Abbildung 5: Benutzeroberfläche des Process Modelers

Zeichenfläche / Editor

In der Mitte befindet sich die Zeichenfläche (der Editor) zum Erstellen der Prozesse. Das Modellieren erinnert an die Arbeit mit einem UML-Modellierungswerkzeug. Die Zeichenfläche ist nicht komplett von abaXX entwickelt worden, sondern setzt auf dem kommerziellen ILOG JViews auf. Hierbei handelt es sich um ein Java-Framework, das als Ausgangsbasis für das Erstellen von grafischen Editoren genutzt werden kann. Auf JViews wird in einem der späteren Kapitel näher eingegangen. Die Zeichenfläche ist eine von JViews zur Verfügung gestellte Komponente (vom Typ `JComponent`). Zwischen den offenen Prozessen kann über Tabs navigiert werden. Die Zeichenfläche ist ein zentraler Bestandteil des Process Modelers, alle Zugriffe auf einen offenen Prozess und seine Elemente erfolgt über den Editor.

Menüleiste

Die Menüleiste erlaubt den Zugriff auf verschiedene Funktionen innerhalb des Process Modelers. Die meisten Aktionen beziehen sich auf den gerade aktiven (sichtbaren) Prozess. Es sind folgende Hauptmenüpunkte definiert:

- File (Erstellen neuer Prozesse; Speichern, Schliessen und Drucken offener Prozesse)
- Edit (Auschneiden, Kopieren und Einfügen von Objekten zwischen den offenen Prozessen)
- Search (Suche von Objekten und Attributen im gerade aktiven Prozess)
- View (Ein- und Ausschalten der Gallerien; Zoomen im gerade aktiven Prozess)
- Code (Generierung von Quellcode; Erstellung fehlender Endpunkte im Prozess)
- Refactor (Extrahieren eines Sub-Prozesses; Konvertieren von Aktivitäten und Prozessen)
- Tools (Export von Prozessen; Deployment von Prozessen; Änderungen am Layout)
- Help

Neben den oben aufgeführten Funktionen stehen noch viele weitere zur Verfügung. Auf die Beschreibung aller einzelnen Menüpunkte wird im folgenden verzichtet. Im Laufe der Entwicklung wurde der Process Modeler um eine Plug-In-Struktur erweitert. Einige Funktionen (zum Beispiel unter Tools) stehen nur dann zur Verfügung, wenn die entsprechenden Plug-Ins installiert sind.

Horizontale Toolbar

Um die Verwendung oft benutzter Funktionen aus der Menüleiste zu erleichtern, sind einige dieser auch über Icons in der Toolbar zu erreichen.

Vertikale Toolbar

Die vertikale Toolbar stellt die zum Modellieren notwendigen Elemente und Werkzeuge zur Verfügung. Es stehen dort unter anderem Aktivitäten, Prozesse und Verbindungen bereit. Die Elemente lassen sich per Drag & Drop auf die Zeichenfläche ziehen. Auch das Umschalten der Werkzeuge erfolgt über die vertikale Toolbar. Zu den Werkzeugen zählen zum Beispiel die Lupe mit der heran- oder herausgezoomt werden kann oder das

Auswahlwerkzeug zum Selektieren von Elementen auf der Zeichenfläche. Die Toolbar lässt sich mit eigenen Elementen aus den Galerien erweitern. Hierzu werden diese einfach per Drag & Drop in die Toolbar gezogen.

Galerien (Repositories)

Die Galerien spielen eine wichtige Rolle im Process Modeler. Sie ermöglichen die Verwaltung der einzelnen Modellierungselemente. Zu diesen zählen Aktivitäten, Prozesse und Parts. Daneben gibt es zwei weitere Galerien mit Business Objekten und Teilnehmern. Wie auch bei der Toolbar lassen sich die Elemente per Drag & Drop auf die Zeichenfläche ziehen. Die Galerien speichern ihre Daten in XML-Dateien und halten diese so persistent. Nach dem Hinzufügen neuer Elemente in eine Galerie muss diese abgespeichert werden, damit die neuen Objekte in die XML-Datei übernommen werden. Die in den Galerien verwalteten Elemente sind immer einem bestimmten Projekt zugeordnet. Neben dem Hinzufügen neuer Elemente können bereits vorhandene Elemente exportiert werden. Je nach Galerie sind noch weitere Aktionen möglich. Mit Hilfe der Galerien wird eine leichte Wiederverwendbarkeit bereits implementierter Geschäftslogik erreicht, wodurch sich geringere Projektaufwände ergeben. In Abhängigkeit vom Kundenprojekt können weitere, unternehmensspezifische Elemente in die Galerien hinzufügt werden.

Eigenschaftseditor (Property Editor)

Im Eigenschaftseditor werden Einzelheiten zum gerade in der Zeichenfläche ausgewählten Element dargestellt. Innerhalb des Editors kann über Tabs zwischen verschiedenen Seiten gewechselt werden. Neben allgemeineren Dingen wie dem Namen und dem Identifier können hier auch die Input- und Output-Parameter sowie die Exits einer Aktivität angesehen werden. Auch Details über die Implementierung sind hier aufgeführt, wie zum Beispiel der komplette Paket-Pfad zur Java-Klasse. Die meisten Werte können editiert werden. Die vorgenommenen Änderungen werden sofort wirksam. Je nach ausgewähltem Element (Aktivität, Sub-Prozess, Part) werden unterschiedliche Attribute im Eigenschaftseditor aufgeführt.

Übersicht / Übersichtskarte

Die Übersichtskarte stellt eine verkleinerte Darstellung des gerade aktiven Prozesses dar. Sie hilft vor allem bei der Navigation in größeren Prozessen.

Nachrichten

In diesem Unterfenster werden alle Ausgaben der Anwendung für den Benutzer sichtbar gemacht. Die für den Benutzer wichtigen Informationen nach Verwendung der Suchfunktion oder der Codegenerierung werden beispielsweise hier angezeigt.

Statusleiste

Die Statusleiste dient der Übermittlung zusätzlicher Informationen an den Benutzer. Er sieht hier zum Beispiel die aktuelle Position des Mauszeigers auf der Zeichenfläche. Auch Hilfestellungen zur gerade aktiven Funktion werden in der Statusleiste eingeblendet. Zusätzlich kann der Anwender hier die Hilfsgitter der Zeichenfläche aktivieren und deaktivieren.

2.2.2 Vorgehen während der Analyse

Die Analyse des Process Modelers gestaltete sich vor allem am Angang relativ schwierig. Die Anwendung ist etwa 2 bis 3 Jahre alt und wurde von mehreren Entwicklern erstellt. Sie ist im Laufe der Zeit immer weiter gewachsen und besteht inzwischen aus hunderten von Klassen in dutzenden von Paketen. Die meisten Klassen besitzen unzählige Abhängigkeiten untereinander. Die Dokumentation der Anwendung beschränkt sich auf die Elemente der Benutzeroberfläche, der Quellcode ist nur sporadisch dokumentiert. Im großen und ganzen standen der Quellcode und die Entwickler der Anwendung zur Verfügung.

Mit Hilfe des Reverse-Engineering wurde deshalb versucht aus dem vorhandenen Quellcode Informationen herauszubekommen. Im ersten Schritt wurden mit Hilfe von Borland Together Klassendiagramme der Anwendung erstellt. In Folge dessen konnte ein Überblick von der Anwendung gewonnen werden. Der zweite Schritt war die Generierung einer

Javadoc-Dokumentaion, um eventuell vorhandene Kommentare im Quellcode herauszuziehen. Letztendlich konnten die meisten Erfahrungen erst bei der genauen Analyse des Quellcodes und der Programmierung des Prototypen gewonnen werden.

2.3 Fazit

Der Process Modeler ist ein Teil des gesamten abaXX.workflow. Er steht relativ am Anfang des gesamten Prozesses und übernimmt die Modellierung der Geschäftsprozesse. Die Anwendung ist im Laufe der Zeit zu einer umfangreichen Applikation mit vielen Funktionen gewachsen. Die saubere Übernahme aller Funktionen in Eclipse wird gewisse Zeit in Anspruch nehmen.

Der Zeitaufwand für eine Integration der Anwendung in Eclipse hängt hauptsächlich von zwei Punkten ab. Zuerst einmal handelt es sich beim Process Modeler um eine Swing-Anwendung. Eclipse stützt sich hingegen auf SWT. Falls eine Übernahme vorhandener Swing-Komponenten in SWT nicht möglich sein sollte, wird die Integration sehr lange dauern, da alle Teile der Benutzeroberfläche umprogrammiert werden müssen. Ein weiteres Problem stellt die enge Verzahnung zwischen dem Process Modeler und JViews dar. Am Anfang der Integration, so lange die Zeichenfläche nicht ersetzt wird und eine Übernahme von Swing-Komponenten in SWT möglich sein sollte, ist dieses Problem noch vernachlässigbar. Falls der Editor aber im späteren Verlauf ausgetauscht werden soll, wird es zu einem größeren Arbeitsaufwand kommen da auch Änderungen am Modell vorgenommen werden müssen.

3 Analyse des Eclipse Frameworks

> *„Programming without an overall architecture or design in mind is like exploring a cave with only a flashlight: You don't know where you've been, you don't know where you're going, and you don't know quite where you are.“*
> *- Danny Thorpe -*

Eclipse verkörpert eine Java-Entwicklungsumgebung, eine Plattform für Tool-Integration und eine Open Source-Gemeinde (vgl. Shavor, 2004, S.37). Für den Java-Entwickler ist Eclipse nicht nur eine Menge von Java-Tools, sondern gleichzeitig auch eine Basis, auf der Tools von anderen Anbietern integriert werden können. Eclipse zwängt nicht in eine proprietäre Lösung, sondern eröffnet dem Entwickler die Möglichkeiten einer globalen Gemeinde mit unterschiedlichen Programmiersprachen, Betriebssystemen und Umgebungen (vgl. Schill, 2005, S. 6). Bei der Entwicklung von Eclipse wurde eine Plattform-orientierte statt eine Tool-orientierte Denkweise angelegt (vgl. Shavor, 2004, S. 15). Eclipse bietet Schnittstellen für die Integration verschiedenster Tools. Alle Arten von Editoren, die in den Entwicklungsprozess involviert sind, können integriert werden um eine einzige integrierte Plattform zur Verfügung zu stellen.

Die 90er Jahre haben ein phänomenales Internet-Wachstum gesehen, sowohl auf der privaten als auch der kommerziellen Seite. Viele neue Technologien wurden entwickelt, vor allem im Bereich Java (vgl. Schill, 2005, S. 5). Dies hatte eine tiefgreifende Wirkung auf die Arten der Anwendungen. eBusiness-Anwendungen verlangten von den Entwicklern, mit vielen unterschiedlichen Arten von Ressourcen zu arbeiten: XML, Java Server Pages (JSP), Java-Code, Grafikdateien, HTML-Dateien, Enterprise Java Beans (EJB) und so weiter. Jede dieser Technologien basierte auf einem anderen Abstraktionsgrad. Diese heterogene Technologielandschaft führte zu einer komplexen Entwicklungsumgebung mit vielen Tools. Mit separaten Tools für jeweils einen Ressourcentyp wurde die ganze Angelegenheit zu einer echten Belastung, sowohl für den einzelnen Entwickler als auch für alle Mitglieder

eines Teams mit ihren unterschiedlichen Tools. Neben dem rasanten Internet-Wachstum mit den vielen neuen Technologien hat sich eine andere durchgreifende Änderung vollzogen. Die Open Source-Bewegung ist aus ihrem Schattendasein hervorgetreten.

An diesem Punkt kommt Eclipse ins Spiel. Das Eclipse-Projekt wurde von IBM, Object Technology International (OTI) und acht anderen Firmen im November 2001 ins Leben gerufen als Reaktion auf die Notwendigkeit, einen neuen Weg zu finden, damit die Entwickler unabhängig voneinander Tools erstellen können, die so zusammenarbeiten, als wären sie Teil eines einzigen integrierten Tool-Sets. Mittlerweile hat sich Eclipse in einer technischen Gemeinschaft in über 100 Ländern ausgebreitet und es gibt Hunderte von Anbietern, die Tools basierend auf Eclipse erstellen. Neben dem eigentlichen Eclipse-Projekt gibt es verschiedene weitere Projekte, die unter der Schirmherrschaft der Eclipse Foundation stehen und Tools für Eclipse entwickeln, wie zum Beispiel das Graphical Editing Framework (GEF). Das Projekt C Development Tools (CDT) illustriert, dass die Plattform nicht Java-spezifisch ist.

Sowohl Eclipse als auch die Plug-Ins sind vollständig in Java implementiert. Als GUI-Framework zur Erstellung der grafischen Oberfläche wird SWT verwendet. SWT basiert ähnlich wie AWT auf den nativen GUI-Komponenten des jeweiligen Betriebssystems und ist daher nicht plattformunabhängig. Eclipse wird aber für 11 verschiedene Systeme und Architekturen bereitgestellt und ist Sprachen-neutral.

Das Rückgrat der Eclipse-Plattform war bis einschließlich zur Version 2.1 eine universelle integrierte Entwicklungsumgebung. Seit Version 3.0 ist Eclipse selbst nur der Kern, der die einzelnen Plug-Ins lädt, die dann die eigentliche Funktionalität zur Verfügung stellen. Diese Funktionalität nennt sich Rich Client Platform (RCP) und basiert auf dem OSGi-Standard. Seit Version 3.1 unterstützt Eclipse auch J2SE 5.0 (vgl. Stal, 2005, S. 9). Weiterhin wurde die Performance optimiert, sowie die Ant-Integration und diverse kleine Dinge, die das Entwicklerleben verbessern.

3.1 Die Eclipse Public License (EPL)

Eclipse wird unter einer zur Open Source Initiative (OSI) kompatiblen Lizenz mit dem Namen Eclipse Public License (EPL) angeboten. Hierdurch ist eine effiziente kommerzielle Nutzung der Software möglich (vgl. Shavor, 2004, S. 33). Der Quellcode ist kostenlos verfügbar und die Software darf weltweit vertrieben werden.

Anders als bei der GPL muss jedoch nicht jedes auf der Software basierende Werk auch unter die EPL gestellt werden. Wenn neue Module hinzugefügt werden, so dürfen diese unter einer anderen – evtl. auch proprietären – Lizenz vertrieben werden. Wenn jedoch ein Modul, welches unter der EPL steht, verändert wird, so muss dieses auch weiterhin unter der EPL vertrieben werden. Somit ist diese Lizenz näher an der LGPL, als an der GPL.

3.2 Konzepte und Aufbau von Eclipse

Die Benutzeroberfläche von Eclipse ist in mehrere Bereiche unterteilt, in denen sich Editoren und Ansichten befinden. Die Ansichten können zwischen den einzelnen Bereichen hin- und hergeschoben werden, so dass die Oberfläche nach belieben zusammengestellt werden kann. Die Bereiche beziehen sich auf Ressourcen des Benutzers im Arbeitsbereich (Workspace). Jede Eclipse-Installation hat zur Laufzeit einen Arbeitsbereich der ihr zugeordnet ist. Diesen kann der Benutzer bei Bedarf ändern. Der Arbeitsbereich ist ein Ordner auf der Festplatte, in dem sich die Dateien des Anwenders befinden. Mit dem JDT können beispielsweise nur Ressourcen angezeigt und modifiziert werden, die sich in diesem Arbeitsbereich befinden (vgl. Shavor, 2004, S. 34). Die Einstellungen innerhalb von Eclipse sind immer einem Arbeitsbereich zugeordnet und werden dort im Verzeichnis .metadata gespeichert. Dies gilt auch für Dinge wie Tasks und Bookmarks. Wenn ein Projekt von Hand aus dem Arbeitsbereich in ein anderes Verzeichnis verschoben und dann in eine Eclipse-Installation eingebunden wird, so gehen leider alle diese Daten verloren.

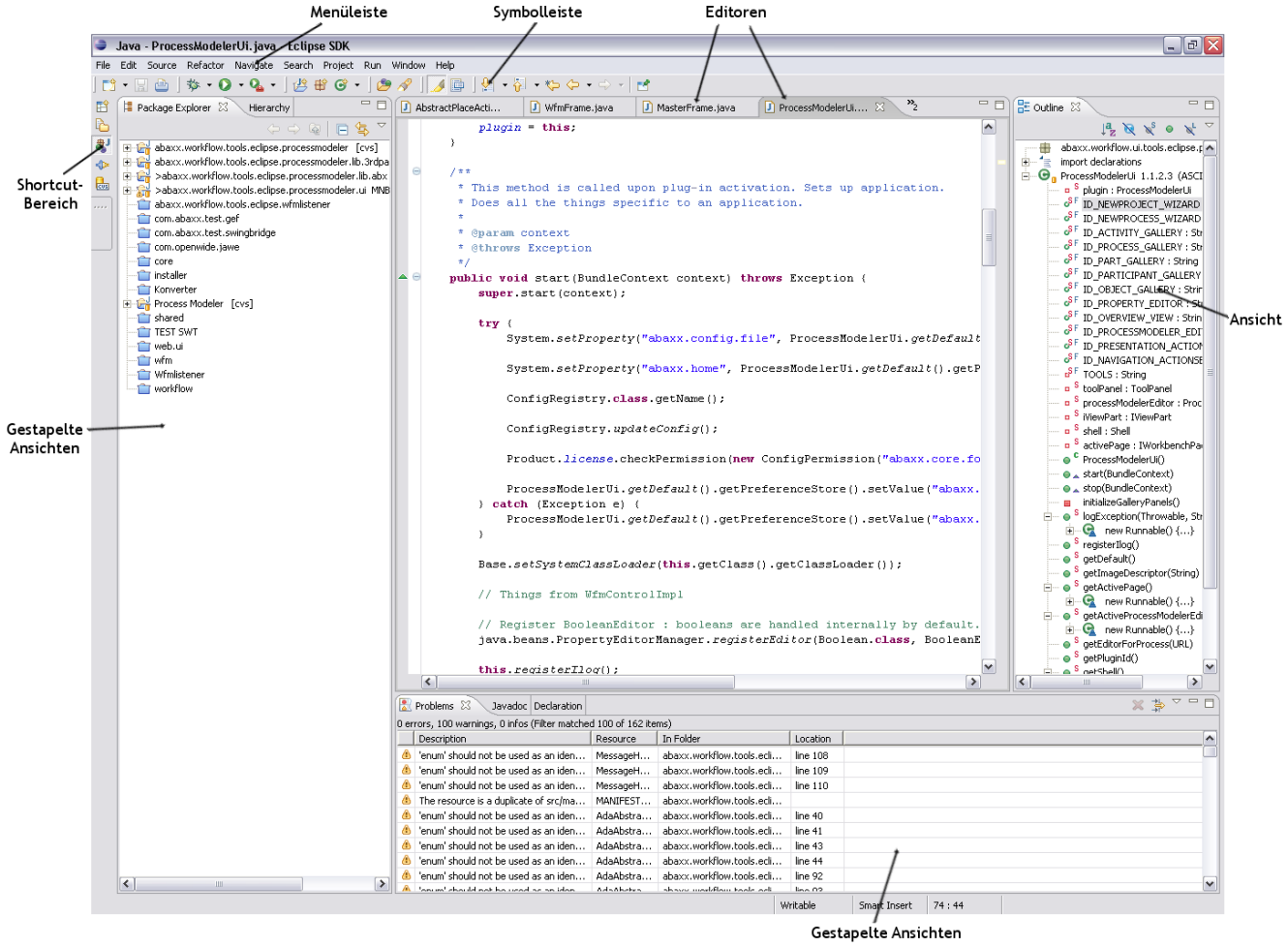

Abbildung 6: Benutzeroberfläche von Eclipse

Eclipse wird, identisch zu anderen Anwendungen, über Elemente wie Menüleisten, Symbolleisten, Editor- und Ansichts-Symbolleisten und Kontextmenüs gesteuert (vgl. Shavor, 2004, S. 45). Die in der Benutzeroberfläche verfügbaren Aktionen hängen von den Plug-Ins ab, die installiert und konfiguriert wurden, sowie der derzeitigen Perspektive. Die gerade aktiven Funktionen hängen von dem ab, was der Benutzer gerade tut. Konkret von der gerade aktiven Ansicht und den dort ausgewählten Elementen.

3.2.1 Plug-Ins und Features

Plug-Ins stellen den zentralen Mechanismus dar, mit dem sich neue Funktionalitäten in Eclipse definieren lassen (vgl. Shavor, 2004, S. 199). Sie stellen die Funktionen bereit, die in einer Eclipse-Konfiguration aufgeführt sind. Doch nicht nur neue Tools bestehen auch

Plug-Ins, auch Eclipse selbst ist aus ihnen aufgebaut. Im Hauptverzeichnis von Eclipse gibt es das Verzeichnis plugins. Dieses Verzeichnis enthält alle Plug-Ins, die in die Eclipse-Installation eingebunden sind. Diese befinden sich entweder in einem eigenen Unterverzeichnis oder als JAR-Datei direkt im Verzeichnis plugins. Der Name eines Plug-Ins setzt sich aus seiner ID und Version zusammen (vgl. Shavor, 2004, S. 180ff.).

Plug-Ins lassen sich zwar durch einfaches Entpacken in das plugins Verzeichnis von Eclipse installieren, sie werden damit aber noch nicht verwaltet. Der Update-Manager von Eclipse verwaltet so genannte Features, unabhängig davon, auf welche Weise diese hinzugefügt wurden. Mit ihnen lassen sich Plug-Ins und andere Features organisieren und strukturieren. Einzelne Plug-Ins werden als Feature verpackt. Auf diese Weise bilden sie einen logischen Container für die Plug-Ins. Weiterhin lassen sich Features ineinander verschachteln. Sie sind eine installierbare Funktionseinheit und lassen sich durch den Update-Manager einfach verwalten, zum Beispiel installieren / deinstallieren oder aktivieren / deaktivieren. Wie bei Plug-Ins gibt es auch für Features ein Verzeichnis features in dem sich alle in die Eclipse-Installation eingebundenen Features befinden. Wie auch bei Plug-Ins besteht der Verzeichnisname eines Features aus seiner ID und Version.

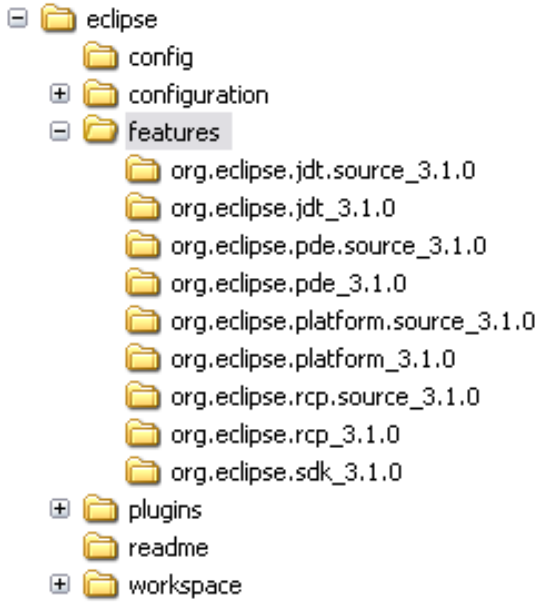

Abbildung 7: Eclipse Verzeichnisstruktur

Die mit dem Update-Manager verwaltete Konfiguration ist ein Satz von Informationen. Eclipse kann daraus beim Startvorgang entnehmen, welche Features eingebunden sind. Bei der Initialisierung werden die Konfigurationsdaten als Metadaten im Arbeitsbereich gespeichert (vgl. Shavor, 2004, S. 191).

3.2.2 Eclipse SDK Plattform

Obwohl Eclipse von Anfang an als eine Integrationsplattform konzipiert war, brauchte es konkrete Anwendungen, die zeigten, was die Plattform wirklich leisten konnte (vgl. Stal, 2005, S. 10). Aus diesem Grund wurde von Anfang an eine integrierte Java Entwicklungsumgebung mit Eclipse ausgeliefert, die Bestandteil der Eclipse Software Development Kit (SDK) Plattform (siehe Abbildung 8) ist und eine Reihe von Werkzeugen mitbringt (vgl. Shavor, 2004, S. 25). Im Lauf der Zeit wurde die Eclipse SDK Plattform um das Plugin Development Environment (PDE) erweitert, das bei der Entwicklung neuer Eclipse Plug-Ins helfen soll. Im Folgenden eine kurze Beschreibung dieser beiden Werkzeuge.

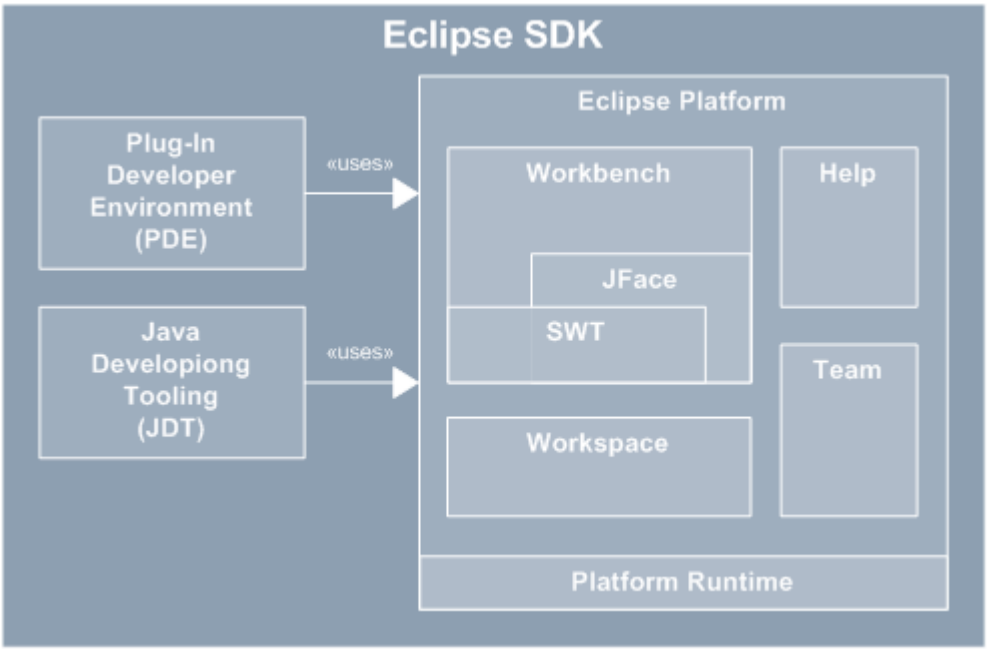

Abbildung 8: Architektur des Eclipse SDK

Java Development Toolkit (JDT)

Das Java Development Toolkit liefert verschiedene Werkzeuge zur Entwicklung von Java Anwendungen (vgl. Schill, 2005, S. 6). Neben einem Editor für Java-Klassen, der unter anderem Syntax-Highlighting unterstützt, werden Assistenten zur Erzeugung von Quellcode und Tools für das Refactoring angeboten. Eines der hilfreichsten Werkzeuge ist aber der Debugger, bei dem sich der Quellcode auch während des Debugens ändern lässt.

Plugin Development Environment (PDE)

Um die Entwicklung von Plug-Ins für Eclipse zu unterstützen, ist das Plug-In Development Environment im Lauf der Zeit Teil der SDK Plattform geworden. Es unterstützt den Entwickler bei der Erstellung, der Entwicklung, dem Testen, dem Debugging und der Auslieferung neuer Plug-Ins. Zusätzlich werden Werkzeuge zur Erstellung von Features und Update Sites zur Verfügung gestellt.

3.2.3 Eclipse RCP Plattform

Während Eclipse in den ersten Versionen eine klare Ausrichtung als Java-IDE hatte und die meisten Entwickler sie hauptsächlich so benutzen, sind die Anwendungsmöglichkeiten von Eclipse mit der Zeit vielfältiger geworden. Bis einschließlich Version 2.1 waren einige Teile der Entwicklungsumgebung fest mit der Eclipse Runtime verwoben (vgl. Schill, 2005, S. 7). Die Entwicklung einer eigenständigen Anwendung, die auf dem Eclipse Framework aufbaut, war ohne die Verwendung von Teilen der IDE unmöglich. Mit Version 3 von Eclipse wurde ein neues Konzept eingeführt (vgl. Schill, 2005, S. 1). Zusammen mit der Community wurde die Plattform verallgemeinert und die Vorteile des Eclipse Kerns wurden in eine universelle, von der IDE unabhängige, Plattform ausgelagert (vgl. Stal, 2005, S. 10). Auf Basis dieser Eclipse Rich Client Platform (RCP) lassen sich Anwendungen erstellen welche die Vorteile der konsequenten Plug-in-Struktur von Eclipse nutzen und von den vielen Features profitieren.

RCP stellt ein Subset der Eclipse Plattform dar. Es enthält die Komponenten SWT und JFace. Weiterhin ist die Eclipse-Runtime enthalten, die den zugrunde liegenden Plug-In-Mechanismus bereitstellt. Darüber angesiedelt ist die (Generic-) Workbench-Komponente, die es unter Benutzung von SWT und JFace erlaubt, die eigene Anwendung mit verschiedenen Sichten, Editoren, Perspektiven, Assistenten, und allem Sonstigen auszustatten, was Eclipse zu bieten hat.

Der Einarbeitungsaufwand ist deutlich höher als bei einfacheren Anwendungen, die sich mit SWT und JFace begnügen. Immerhin gibt es mittlerweile ein Plug-In, das Entwicklern eine Menge Arbeit bei der Erstellung solcher Rich Clients abnimmt.

3.3 Erweiterbarkeit

Die Eclipse-SDK-Konfiguration unterstützt standardmäßig die Java-Technologie, das eigentliche Ziel von Eclipse besteht aber darin, eine allgemeine Entwicklungsumgebung und Tool-Integrationsplattform bereitzustellen, die von verschiedensten Leuten verwendet und erweitert werden kann (vgl. Shavor, 2004, S. 199ff.). Um dies zu erreichen ist es notwendig, dass fast alle Aspekte der Umgebung angepasst werden können und neue Funktionalität von unterschiedlichen Softwareanbietern aufgenommen werden kann und dabei das Bild einer einheitlichen Umgebung gewahrt bleibt (vgl. Shavor, 2004, S. 73). Eine allgemeine Benutzeroberfläche ist aber mehr als nur gleich aussehende Menüs und Schaltflächen. Die gesamte Interaktion des Benutzers mit der Anwendung muss bei allen Plug-Ins gleich sein.

Eclipse bietet hierzu ein Komponentenmodell mit Plug-Ins und einem Erweiterungsmechanismus, der es einem Plug-In ermöglicht neue Fukitonen mit Erweiterungen und Erweiterungspunkten einzuführen (vgl. Stal, 2005, S. 10). Ein Erweiterungspunkt definiert prinzipiell die Stelle innerhalb der Benutzeroberfläche, an der ein hinzugefügtes Plug-In eine neue Funktionalität festlegen kann. Hierdurch wird ein Architekturmechanismus eingeführt, der die Erweiterbarkeit öffentlich deklariert. Durch

die eindeutige Trennung der Deklaration eines Elements in einer Manifest-Datei von ihrem Implementierungscode in einer JAR-Datei, kann das Element erst bei Bedarf und so spät wie möglich initialisiert werden. Damit kann die Erweiterung in der Workbench erscheinen, selbst wenn das Plug-In noch nicht geladen ist. Dies ist der Grund dafür, dass Eclipse auch mit sehr vielen Plug-Ins noch performant ist.

Die Definition in der Manifest-Datei legt fest, wo und wann die Erweiterung erscheint und wie sich sich für den Benutzer präsentiert (vgl. Shavor, 2004, S. 238ff.). Im Code wird die Operation definiert, die auszuführen ist, wenn der Benutzer die Erweiterung verwendet.

Abbildung 9: Aktionserweiterungen in der Symbolleiste

Im folgenden ein einfaches Beispiel für die Deklaration einer Aktionserweiterung. Die in Abbildung 9 aufgeführten Erweiterungen würden in der plugin.xml jeweils mit der in Listing 1 dargestellten Definition deklariert.

```
<action
      id=''abaxx.workflow.tools.eclipse.processmodeler.ui.fitonscreenaction''
      label=''%plugin.actions.centerzoom.label''
      tooltip=''%plugin.actions.centerzoom.tooltip''
      icon=''icons/action_centerzoom.gif''
      toolbarPath=''org.eclipse.ui.edit.text.actionSet.presentation/Presentation''
      style=''push''
      class=''abaxx.workflow.ui.tools.eclipse.processmodeler.ui.actions.FitOnScreenAction''
/>
```

Listing 1: Deklaration einer Aktionserweiterung in der plugin.xml

Jede Erweiterung benötigt eine `id` durch die sie Eclipse weit eindeutig referenziert werden kann. Der unter `label` eingegebene Text erscheint für den Benutzer sichtbar im Menü oder als Beschriftung eines Schaltknopfs, falls kein Icon verwendet wird. In diesem Fall, da es

sich um einen Schaltknopf handelt, bewirkt der eingegebene Text nichts. Wenn der Benutzer mit der Maus über den Schaltknopf fährt wird der unter `tooltip` eingegebene Text angezeigt. Beide Werte verweisen in diesem Beispiel auf Einträge in einer Property-Datei. Auf diese Weise kann die Anwendung einfach internationalisiert werden. Die Grafik des Schaltknopfs wird über `icon` relativ zur plugin.xml referenziert. Der `toolbarPath` gibt die Position an, an der die Grafik in der Benutzeroberfläche erscheinen soll. Dabei wird auf IDs anderer Erweiterungen verwiesen. Über `style` kann der Typ des Schaltknopfs beeinflusst werden. In diesem Fall handelt es sich um einen Knopf welchen nach dem Drücken wieder zurückkommt. Andere bleiben zum Beispiel gedrückt. Der letzte übergebene Wert `class` gibt die Klasse an, die diese Erweiterung implementiert. Im Fall der hier beschriebenen Aktionserweiterungen wird die dortige `run()`-Methode aufgerufen. Die Klasse muss eine Schnittstelle implementieren, die die `run()`-Methode verlangt.

3.3.1 Aktionserweiterungen

Aktionserweiterungen repräsentieren Aktionen, die dem Benutzer in Menüs und Symbolleisten zur Verfügung stehen (vgl. Shavor, 2004, S. 237). Neben dem Hauptmenü und der Hauptsymbolleiste können sie auch in den Menüs von Ansichten und Editoren erscheinen, wie auch in den verschiedenen Kontextmenüs. Aktionen werden in so genannten ActionSets gruppiert, um sie besser strukturieren zu können. Unter WINDOW -> CUSTOMIZE PERSPECTIVE... sind alle ActionSets aufgeführt, die in einer Eclipse-Installation existieren.

3.3.2 Dialogfelder und Assistenten

Dialogfelder und Assistenten werden immer wieder gebraucht, sei es bei der Erstellung neuer Ressourcen als Erweiterung der Eclipse-Benutzerobrefläche oder zur Anzeige von Fehlermeldungen als Allzweckdialogfelder (vgl. Shavor, 2004, S. 279). Auch Grundeinstellungs- und Eigenschaftsseiten fallen in diese Kategorie. Eclipse bietet eine große Auswahl fertiger Dialogfelder und Assistenten, wiederverwendet werden können um

die Gesamtfunktionalität und die Integration in die Benutzeroberfläche zu verbessern. Auf diese Weise kann das Tool produktiver entwickelt werden. Auch die Einheitlichkeit im Aussehen der einzelnen Tools wird durch die Wiederverwendung gefördert.

3.3.3 Ansichten

Mit Hilfe von Ansichten kann durch Ressourcen oder andere Informationen navigiert werden (vgl. Shavor, 2004, S. 34). Im Vergleich zu Editoren erscheinen Modifikationen in Ansichten sofort, ohne dass diese erst gespeichert werden müssen. Mit Ansichten lässt sich vieles anstellen: sie können in der Größe geändert, übereinander gestapelt und auf der Shortcut-Leiste angeordnet werden. Ihr hauptsächlicher Zweck ist die Bereitstellung von zusätzlichen Informationen zur gerade im Editor offenen Ressource. Es ist immer nur eine Instanz einer bestimmten Ansicht verfügbar.

3.3.4 Editoren

Neben Ansichten und Perspektiven sind Editoren die grundlegenden Bausteine der Benutzeroberfläche von Eclipse (vgl. Shavor, 2004, S. 351). Editoren werden im Allgemeinen verwendet, um Ressourcen zu modifizieren (vgl. Shavor, 2004, S. 34). Jeder Editor hat eine Anzahl von Ressourcen denen er zugeordnet ist.

Eclipse erleichtert die Erstellung von Editoren mit einem Editor-Framework. Es deckt die allgemeinen Verhaltensweisen eines Editors ab und der Entwickler kann sich auf die eigentliche Funktionalität konzentrieren. Mit Editoren lassen sich nicht nur Textdateien bearbeiten. Wie im Fall des Process Modelers gibt es zum Beispiel auch grafische Editoren. Andere wiederum haben mehrere Seiten, um die Darstellung von komplexen Dateien besser strukturieren zu können. Eclipse reserviert einen Editorbereich für alle offenen Editoren, meist ist dieser in der Mitte der Benutzeroberfläche angesiedelt. Im Unterschied zu Ansichten können beliebig viele Instanzen eines Editors offen sein.

3.3.5 Perspektiven

Mit Hilfe einer Perspektive lässt sich in Eclipse die Benutzeroberfläche in einer bestimmten Art und Weise konfigurieren (vgl. Shavor, 2004, S. 34). So kann beispielsweise definiert werden, welche Ansichten angezeigt werden sollen und an welcher Stelle. Es kann auch definiert werden, ob bestimmte Aktionen angezeigt werden sollen oder nicht. Eclipse kann mehrere Perspektiven gleichzeitig öffnen, wobei nur eine aktiv sein kann. Mit Hilfe von Perspektiven kann die Oberfläche für verschiedene Nutzungsszenarien konfiguriert werden.

3.3.6 Ressourcen

Standardmäßig hat jede installierte Eclipse-Kopie einen Arbeitsbereich, um Projekte, deren Ressourcen und die dazu gehörenden Informationen zu speichern. Ein Projekt ist eine Art Container, der andere Ressourcen aufnimmt (vgl. Shavor, 2004, S. 35f.). In Projekten werden auch zusätzliche Informationen über die in ihnen enthaltenen Ressourcen gespeichert, wie zum Beispiel Angaben zum Klassenpfad. Einem Projekt können Naturen und Builder zugeordnet werden. Mit Hilfe einer Natur wird einem Projekt Verhalten und Funktion beigebracht (vgl. Shavor, 2004, S. 407). Ein Builder erlaubt es, geänderte Dateien zu bestimmten Zeitpunkten zu verarbeiten. Hierdurch kann die inkrementelle Entwicklung unterstützt werden.

3.3.7 Hilfe

Eclipse bietet Mechanismen an, mit deren Hilfe die eigene Dokumentation in das Hilfe-Framework von Eclipse integriert werden kann. Der Inhalt kann ein Satz von HTML-Dateien oder eine PDF-Datei sein (vgl. Shavor, 2004, S. 515). Neben einer Suchfunktion innerhalb der Dokumente lässt sich auch eine kontextabhängige Hilfe implementieren, die beim Markieren bestimmter Elemente in der Benutzeroberfläche das zu diesem Element passende Hilfethema anzeigt.

3.4 Fazit

Eclipse ist in den letzten Jahren immer beliebter geworden. Eclipse verspricht Zeit und Geld zu sparen, da alle Team-Mitglieder mit der gleichen Entwicklungsumgebung arbeiten (vgl. Schill, 2005, S. 6f.). Da Eclipse ein Open Source-Projekt ist kann zusätzlich der gesamte Quellcode eingesehen und verändert werden und zwar kostenlos (vgl. Shavor, 2004, S. 25). Zusätzlich verbessert die große Entwicklergemeinde Eclipse ständig.

Eclipse ist komplett aus Plug-Ins aufgebaut. Die Plug-In-Architektur von Eclipse ist das wichtigste Merkmal. Seine umfangreichen Java-Frameworks stellen die wiederverwendbaren Komponenten bereit, die Tool-Anbieter in eigene Tools integrieren oder mit denen der Endbenutzer seine Eclipse-Umgebung erweitern kann (vgl. Shavor, 2004, S. 37). Auf diese Weise ist eine Konzentration auf die letzten 20% des Codes möglich, welche die eigentliche Funktionalität des Tools implementieren. Derselbe Code muss nicht erneut geschrieben werden, wie es in der Vergangenheit oft der Fall war.

Mit Hilfe verschiedener Java-Frameworks läuft das Erweitern von Eclipse schneller ab und sie fördern zusätzlich das einheitliche Aussehen der Benutzeroberfläche. Zusätzlich ist die Oberfläche näher am System, da Eclipse nicht wie traditionelle Java-Anwendungen AWT oder Swing sondern das für diese Zwecke entwickelte SWT verwendet. Letztlich erhält der Endbenutzer dadurch eine produktivere und geschlossenere Umgebung und kann sich auf seine Arbeit konzentrieren (vgl. Shavor, 2004, S. 199).

Durch die Verwendung von System nahen GUI-Komponenten und die Verwendung zahlreicher Entwurfsmuster welche die Performance steigern, wie zum Beispiel Lazy Loading, ist Eclipse in der Lage schneller zu arbeiten als andere Java-Anwendungen.

Eclipse ist nicht wirklich schwer. Es erfordert jedoch eine gewisse Zeit sich in das Framework einzuarbeiten und alle zur Verfügung stehenden Möglichkeiten kennenzulernen.

Für die Aufgabenstellung innerhalb der Diplomarbeit ist die Eclipse SDK Plattform besser geeignet als die Eclipse RCP Plattform, da sie bereits essentielle Bestandteile einer Entwicklungsumgebung zur Verfügung stellt auf denen aufgebaut werden kann. Als Grundlage der weiteren Arbeit wird deshalb die Eclipse SDK Plattform angenommen.

4 Technologien

„Ohne Unterschied macht Gleichheit keinen Spaß.“

- Dieter Hildebrandt -

4.1 Grafische Bibliotheken

Im folgenden werden die grafischen Bibliotheken AWT / Swing und SWT zur Erzeugung und Darstellung grafischer Benutzerschnittstellen (GUI) kurz vorgestellt und miteinander verglichen. Da der derzeitige Process Modeler auf AWT / Swing aufbaut und das zukünftige Eclipse Plug-In in SWT implementiert sein wird, ist es für die Diplomarbeit wichtig zu verstehen, wo die Vor- und Nachteile der einzelnen Bibliotheken liegen und welche Unterschiede zwischen ihnen bestehen.

4.1.1 Abstract Window Toolkit (AWT) / Swing

Der erste Ansatz zur GUI-Entwicklung mit Java war das plattformunabhängige Abstract Window Toolkit (AWT). Es stellt so genannte schwergewichtige (heavyweight) Komponenten zur Darstellung von Steuerelementen zur Verfügung. Schwergewichtig bedeutet, dass die nativen Widgets des jeweiligen Hostsystems zur Darstellung verwendet werden. Um trotz schwergewichtiger Komponenten plattformunabhängig zu bleiben, ist beim AWT der Java-basierte Teil der Widgets über alle Plattformen identisch und nur der in C implementiere Teil kapselt die Unterschiede. Da die meisten neueren Anwendungen inzwischen das auf AWT aufbauende Swing verwenden und AWT als veraltet gilt, konzentrieren sich die folgenden Ausführungen auf Swing.

Swing, die Nachfolge-Bibliothek des AWT, ist ein von Sun Microsystems entwickeltes Toolkit zur Programmierung grafischer Oberflächen, das seit der Java-Version 1.2 Bestandteil der Java-Runtime ist (vgl. Ullenboom3, 2006). Swing gehört zu den Java Foundation Classes

(JFC), die eine Sammlung von Bibliotheken zur Programmierung grafischer Benutzerschnittstellen bereitstellen. Zu diesen Bibliotheken gehören Java2D, die Accessibility-API, die Drag & Drop-API und das AWT.

Swing setzt teilweise auf AWT auf, implementiert die bereitgestellten Komponenten jedoch vollständig in Java (vgl. Ullenboom1, 2006). Dies bedeutet, dass die Komponenten direkt von Java gerendert werden und nicht von nativen Betriebssystemkomponenten abhängen. Sie werden deshalb als leichtgewichtige (leightweight) Komponenten bezeichnet und sind plattformunabhängig. Der Nachteil ist, dass eine Swing-Anwendung nicht wie eine für das Betriebssystem entwickelte Anwendung aussieht. Dieses Problem kann aber mit entsprechenden Look & Feels kompensiert werden. Ein weiteres Manko von Swing ist die schlechte Performance, bedingt durch das Rendern der Komponenten durch Java. Diese hat sich in den letzten Jahren aber durch verbesserte Hardwareunterstützung der Beschleunigungsfunktionen von Grafikkarten. Eine Mischung von AWT- und Swing-Komponenten ist nicht zu empfehlen, da schwergewichtige Komponenten vom Betriebssystem gezeichnet und dadurch über die leichtgewichtigen gelegt werden.

```
public class HelloWorldSwing {

      public static void main(String[] args) {
            JFrame jFrame = new Jframe(''HelloWorldSwing'');

            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            JLabel jLabel = new Jlabel(''Hello World!'');

            frame.getContentPane().add(jLabel);

            frame.pack();

            frame.setVisible(true);
      }

}
```

Listing 2: HelloWorld in Swing

Jede grafische Anwendung fängt mit einem einfachen Fenster an (vgl. Ullenboom2, 2006). Die Fenster von Swing unterscheiden sich nicht stark von den AWT-Fenstern. Statt `Frame` heißt die Klasse bei Swing JFrame. Nach dem Erzeugen verfügt das Fenster über keine Funktionalität, selbst das Schliessen muss erst implementiert werden. Hierzu wird die Funktion `setDefaultCloseOperation()` verwendet und dieser das Argument `Jframe.EXIT_ON_CLOSE` übergeben. Mit der Methode `setVisible(true)` wird das Fenster letztlich angezeigt. Bei Swing nimmt ein Container Komponenten auf und setzt sie mit Hilfe eines Layoutmanagers in die richtige Position. `JFrame` ist ein solcher Container. Komponenten werden mit Hilfe der `add()`-Methode einem Container hinzugefügt.

4.1.2 Standard Widget Toolkit (SWT) / JFace

Das Standard Widget Toolkit (SWT) und das darauf basierende JFace stellen eine von IBM entwickelte Alternative zu AWT und Swing (vgl. Rutishauser, 2003, S. 1) da. SWT und JFace befähigen Entwickler, Benutzeroberflächen von der Drei-Button-Anwendung in SWT bis hin zu RCP-basierten Multiwindow-Anwendung zu erarbeiten.

Im Gegensatz zu Swing schlägt SWT den gleichen Weg wie AWT ein: das Ausnutzen der nativen Widgets auf dem jeweiligen Hostsystem. SWT ist dabei nur etwas geschickter. Bei SWT unterscheidet sich selbst die Implementierung auf Java-Ebene schon von System zu System, wobei das Interface nach außen identisch ist. Ein weiterer wichtiger Unterschied liegt darin, dass SWT gewissermaßen eine Obermenge aller Widgets zur Verfügung stellt. Wenn bestimmte Widgets auf einer Plattform nativ nicht vorhanden sind, wie Toolbars unter Motif, emuliert das SWT sie. Diese Widgets sind dann im Gegensatz zu den ansonsten schwergewichtigen Widgets leichtgewichtig (ähnlich wie bei Swing).

Dem Entwickler nötigt SWT eine gewisse Umgewöhnung ab, was den allzu sorglosen Umgang mit Ressourcen angeht. Den im Gegensatz zur sonstigen Herangehensweise, das Freigeben nicht genutzter Objekte dem Garbage Collector zu überlassen, zeichnet der Entwickler hier für deren explizites Freigeben verantwortlich (vgl. Rutishauser, 2003, S. 2). Das geplante

Ableben jeder SWT-Komponente ist durch den Aufruf der *dispose*-Methode zu besiegeln (ähnlich den Methoden *free* und *delete* bei C/C++). Dabei gilt allerdings, dass beim *dispose* eines Container-Widgets alle darin enthaltenen Widgets auch mit *dispose* freigegeben werden.

Im folgenden das viel bemühte HelloWorld in SWT. Im Gegensatz zu Swing fällt die kurze Startup-Zeit beim Ausführen auf, subjektiv erscheint es wesentlich flinker als unter Swing.

```
public class HelloWorldSWT {

	public static void main(String[] args) {
		Display display = new Display();

		Shell shell = new Shell(display);

		Label label = new Label(shell, SWT.NONE);

		label.setText(''Hello World!'');

		label.setBounds(10, 10, 100, 100);

		shell.pack();

		shell.open();

		while (!shell.isDisposed()) {
			if (!display.readAndDispatch())  {
				display.sleep();
			}
		}

		display.dispose();

		shell.dispose();
	}

}
```

Listing 3: HelloWorld in SWT

Zwei wesentliche Komponenten, die in jedem SWT-Programm vorhanden sein müssen, sind `Shell` und `Display` (vgl. Rutishauser, 2003, S. 4). Ein `Display`-Objekt stellt die Brücke zum darunter liegenden Betriebssystem bereit und bildet die SWT- / JFace-Welt in das jeweilige OS ab. Außerdem stellt `Display` das Event-Handling sicher. Im Regelfall dürfte jede Anwendung genau ein `Display` besitzen, wobei dieses keine visuelle Repräsentation hat. Letzteres stellt eine `Shell` bereit, die für das eigentliche Fenster einer Anwendung verantwortlich ist und die im weiteren Verlauf erstellten Objekte aufnimmt.

Im Unterschied zu Swing bekommen neue Komponenten schon bei ihrer Erstellung das übergeordnete Parent-Objekt übergeben. Desweiteren wird ein Style-Parameter übergeben. Dieser definiert im Wesentlichen die Ausprägung beziehungsweise das Aussehen des Widget. Statt eines PushButton kann durch Angabe von `SWT.CHECK` oder `SWT.RADIO` ein CheckButton beziehungsweise RadioButton erzeugt werden. `SWT` ist die Klasse, die die entsprechenden Konstantendefinitionen bereit hält. Kombinationen verschiedener Stile bewirken eine (bitweise) Oder-Verknüpfung. Um demnach einen links ausgerichteten Button zu erzeugen, müsste es `SWT.PUSH | SWT.LEFT` heißen. Der Style-Parameter kann nachträglich nicht mehr verändert werden.

Ein weiterer Unterschied zu Swing ist die Endlosschleife, die nach dem Öffnen der `Shell` aufgerufen wird. Sie läuft, so lange das Fenster nicht geschlossen wird. Bei jedem Durchlauf werden die anstehenden Ereignisse in der Event-Warteschlange abgearbeitet, zum Beispiel Klicks auf Buttons oder Eingaben in Textfeldern. Bei Swing passiert natürlich das gleiche im Hintergrund, der Vorgang muss aber nicht explizit vom Programmierer aufgerufen werden.

Im Rahmen des Eclipse-Projekts wurde über das SWT hinaus das JFace-Toolkit entwickelt. Es setzt auf SWT auf und erweitert dieses, soll es aber bewusst nicht verdecken. Es bietet dem Entwickler vor allem Hilfestellungen bei den sich mehr oder weniger wiederholenden Aufgaben in der GUI-Entwicklung. Hier finden sich beispielsweise Klassen für die Erstellung von Assistenten oder Standarddialogen, darüber hinaus bekommt man mit JFace ein

modellgetriebenes Werkzeug an die Hand, im Gegensatz zu SWT bei dem die Widgets nur auf direktem Wege zu manipulieren sind. Durch Anwendung des Model-View-Controller-(MVC) Konzepts genügt es bei JFace, das entsprechende Model zu verändern.

4.1.3 Fazit

SWT ist durch seine schwergewichtigen Komponenten performanter als Swing. Dieses konnte aber in den letzten Jahren aufholen was die Performance angeht. Im Vergleich dazu ist Swing mittlerweile ausgereift und es lassen sich ansprechende und funktionelle Anwendungen mit diesem Toolkit bauen. Durch die zunehmende Verbreitung von Eclipse bekommt auch SWT einen zusätzlichen Schub in der Entwicklung. Es gibt immer mehr fertige Widgets auf die ein Entwickler zugreifen kann.

Bei Eclipse erweist sich die JFace-Erweiterung von SWT als sehr hilfreich. Sie bietet dem Entwickler viele vorgefertigte Dialoge und Assistenten auf denen aufgebaut werden kann und erspart somit eine Menge Arbeit. Außerdem gewährleistet sie eine konsistente Benutzeroberfläche.

Einer der größten Nachteile von SWT ist seine Plattformabhängigkeit. Für jede Plattform muss eine speziell angepasste Bibliothek verwendet werden. Diese gibt es aber inzwischen für die wichtigsten Betriebssysteme. Aber gerade die Abhängigkeit von der Architektur ermöglicht die oben erwähnten Performancevorteile.

4.2 Grafische Frameworks

Bei der Erstellung von visuellen und interaktiven Anwendungen werden komplexe Objektmodelle implementiert und untere Grafikschichten, die von der gewählten Plattform zur Verfügung gestellt werden, verwendet (vgl. Joubert, 2005, S. 18). Unter Swing sind schon seit längerem Grafik-Frameworks verfügbar, welche die wichtigsten Funktionen abstrahieren und dem Entwickler in einfacher Form zur Verfügung stellen. Eines von diesen

ist das kommerzielle ILOG JViews. Unter SWT gibt es zur Zeit noch wenige solcher Frameworks. Im Rahmen des Eclipse Projekts wurde speziell für diese Aufgaben das Graphical Editing Framework (GEF) entwickelt. Im folgenden werden beide Frameworks genauer analysiert und miteinander verglichen.

4.2.1 ILOG JViews

ILOG JViews ist ein 2D-Grafik-Paket zur Erstellung von grafischen Benutzeroberflächen (vgl. ILOG, 2002, S. 1ff.). Es erweitert die einfachen Komponenten von AWT und Swing und ermöglicht Java GUI Entwicklern die Erstellung optisch hochwertiger Benutzeroberflächen. Workflow- und Prozessfluss-Diagramme und jedwede Art von Anwendungen, die Karten darstellen, gehören zu solchen Oberflächen.

Der Process Modeler baut zur Zeit auf diesem graphischen Framework auf. Die komplette Zeichenfläche wurde mit Hilfe von JViews implementiert. Auch das Modell der Anwendung stützt sich größtenteils auf Klassen von JViews von denen es erbt. Im Process Modeler werden insgesamt folgende Komponenten der ILOG JViews Component Suite verwendet:

- ILOG JViews Graphics Framework, das Fundament in der ILOG JViews Component Suite
- ILOG JViews Graph Layout, liefert spezielle Unterstützung für alle Anwendungen die Graphen (Netzwerke) aus Knoten und Kanten darstellen
- ILOG JViews Stylable Data Mapper (SDM), führen die MVC-Architektur in gewöhnliche ILOG JViews Oberflächen ein. Dies erlaubt Entwicklern schnell viele ILOG JViews Obeflächen zu bauen und gleichzeitig die volle Kontrolle über deren Anpassbarkeit zu behalten. Eine spezielle Version der SDM zielt auf die Entwicklung von Workflow-Anwendungen ab.

Neben den hier genannten Komponenten gibt es noch weitere wie zum Beispiel das Application Framework, das einen einfachen GUI Editor zum Einstieg mitbringt. Da der Process Modeler diese aber nicht verwendet wird hier nicht näher auf sie eingegangen.

Das ILOG JViews Graphics Framework liefert eine Reihe von JavaBeans, eine API für Entwickler und einen beispielhaften grafischen Editor. Die JavaBeans können in der eigenen Anwendung verwendet werden. Der Process Modeler baut auf solch einer JavaBean auf. Falls die Funktionalität der JavaBeans nicht ausreicht kann die API für Entwickler dazu genutzt werden um diese hinzuzufügen. Die API ist sehr gut dokumentiert und existieren viele Beispiele, welche die Verwendung aufzeigen.

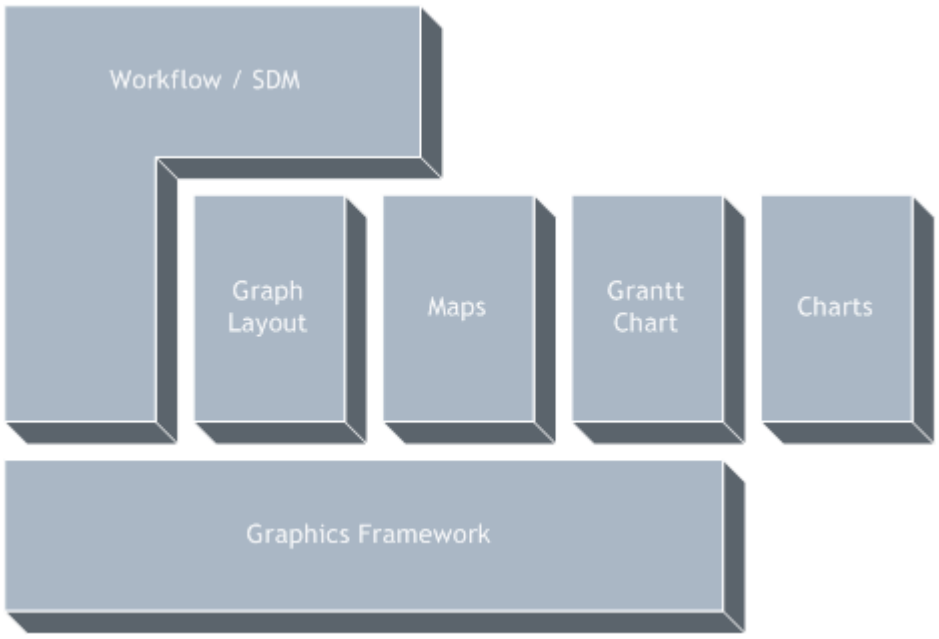

Abbildung 10: ILOG JViews Suite

Architektur des Graphics Framework

Das Graphics Framework von JViews baut auf folgenden drei Teilen auf:

- Grafische Objekte
- Eine Datenstruktur zur Verwaltung der grafischen Objekte
- Eine Zeichenfläche auf der die grafischen Objekte dargestellt werden

Bei ILOG JViews gibt es für jeden oben beschriebenen Teil ein eigenes Objekt.

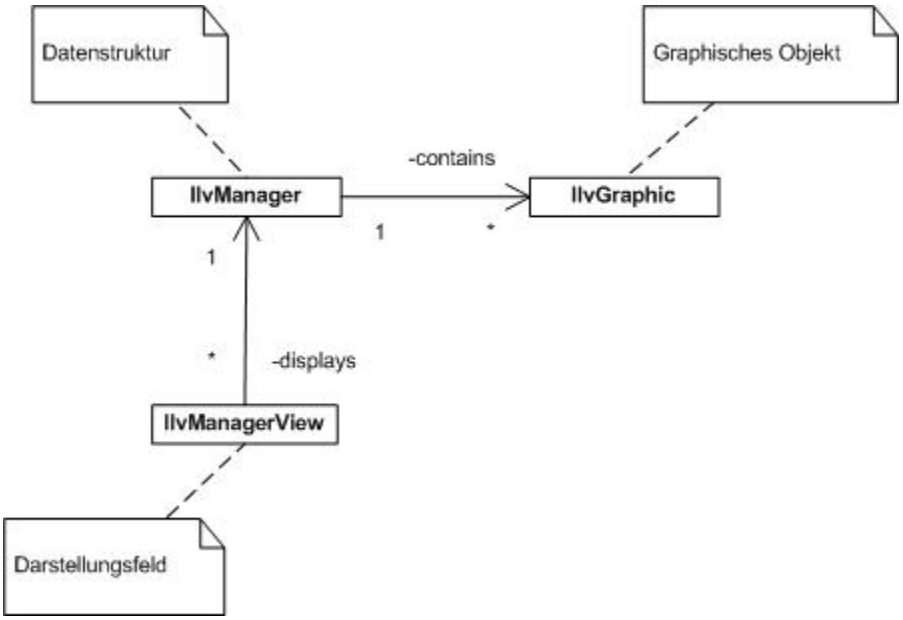

Abbildung 11: Die wichtigsten ILOG JViews Klassen

Das graphische Objekt: `IlvGraphic`

`IlvGraphic` repräsentiert typischerweise ein Objekt, das vom Benutzer gesehen und manipuliert werden kann. In einem Prozessdiagramm könnte es sich hierbei um eine Aktivität oder Kante zwischen Aktivitäten handeln.

Die Datenstruktur: `IlvManager`

`IlvManager` verwaltet die grafischen Objekte und ist das wichtigste Objekt im Graphics Framework. Es ist eine Art Container. Die grafischen Objekte werden in mehreren Ebenen organisiert, wobei Objekte auf höheren Ebenen vor Objekten auf niedrigeren Ebenen gezeichnet werden. Vor bedeutet hierbei über, Objekte auf höheren Ebenen verdecken demnack Objekte auf niedrigeren Ebenen.

Der Viewport: `IlvManagerView`

`IlvManagerView` ist die Zeichenfläche auf der die grafischen Objekte dargestellt werden. Es handelt sich bei ihr um eine Komponente (Subklasse von `java.awt.Component`), die in einer AWT- oder Swing-Anwendung benutzt wird um den Inhalt des Managers anzuzeigen. Da es eine Komponente ist, kann sie so gut wie an beliebiger Stelle eingebunden werden.

4.2.2 Graphical Editing Framework (GEF) / Draw2D

Durch die Verwendung von nativen GUI-Komponenten eignet sich SWT zur Darstellung üblicher grafischer Objekte, wie Buttons und Menüs (vgl. Joubert, 2005, S. 18). Sowohl die Performance als auch die Qualität sind vorzüglich. Bei grafikintensiven Anforderungen stoß SWT aber immer wieder an seine Grenzen, da bisher kein performantes graphisches Framework verfügbar war. Das im Rahmen eines Eclipse Technologie Projekts entwickelte Graphical Editing Framework (GEF) soll genau diese Schwachstelle von SWT ausbügeln. Es erlaubt Entwicklern, ein vorhandenes Anwendungsmodell zu nehmen und zur Darstellung dessen einen graphischen Editor zu entwickeln. Die hierfür verwendete grafische Umgebung ist das auf SWT aufbauende Plugin Draw2D (welches Bestandteil der GEF Komponente ist). Es wird verwendet um die grafischen Objekte zu zeichnen und auszurichten (vgl. GEF, 2005). GEF setzt zusätzlich ein Model-View-Controller (MVC) Framework zum Editieren der Objekte drüber. Dieses Framework gewährleistet, dass alle graphischen Anwendungen in Eclipse ähnlich aussehen und sich gleich verhalten. Der Entwickler kann sich die vielen allgemeinen Funktionen des GEF zu Nutze machen oder diese für seinen speziellen Fall erweitern.

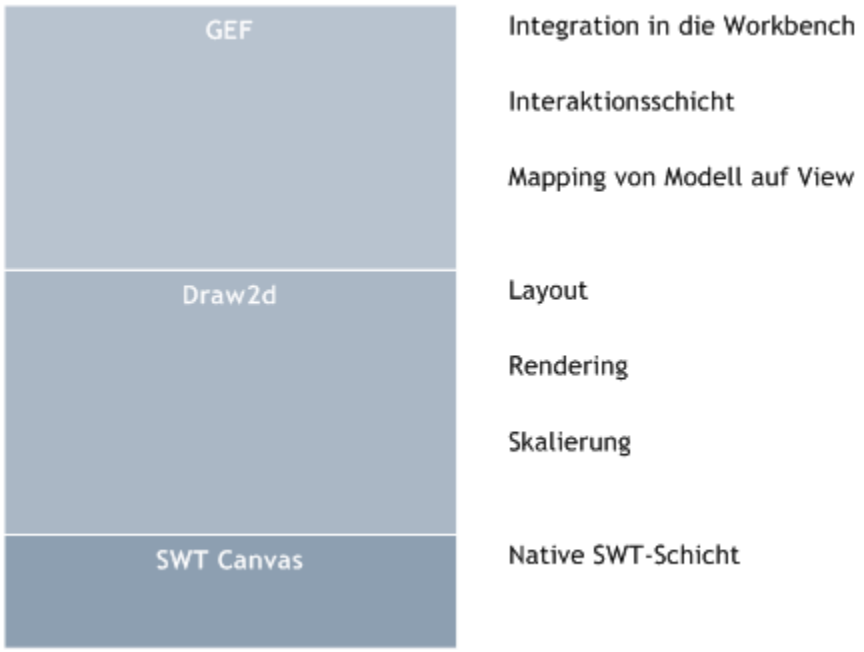

Abbildung 12: Schichten des Graphical Editing Frameworks (GEF)

Das GEF existiert seit etwa fünf Jahren (vgl. Bokowski, 2005, S. 5). Es stammt ursprünglich aus einem IBM Projekt wo es die Basis für grafische Editoren in IBM-Produkten bildete. Es findet zunehmend Verwendung in Open Source- und kommerziellen Produkten. Das GEF eignet sich zur Erstellung einer großen Vielfalt an Anwendungen wie zum Beispiel: Anwendungen zur Erstellung von Ablaufdiagrammen und Benutzeroberflächen, UML Editoren (Workflow- oder Klassen-Diagramme) oder sogar für WYSIWYG-Editoren. GEF bietet unter anderem:

- Werkzeuge zur Auswahl, Erstellung und Verbindung von Objekten
- Eine Palette zur Darstellung dieser Werkzeuge
- Helfer für die Skalierung der Objekte und Ausrichtung und Krümmung von Verbindungen
- Zwei GEF Betrachter: Graphical und Tree
- Ein Controller Framework zum Abbilden des Business Models auf die View
- Undo / Redo Support via Commands und einem CammandStack

Verwendete Architektur- und Entwurfsmuster

Das Graphical Editing Framework verwendet eine Reihe von Architektur- und Entwurfsmustern, die eine bessere Strukturierung der Anwendung mit sich bringen und dem Programmierer das Leben einfacher machen sollen. Im Folgenden eine kurze Beschreibung der wichtigsten. Das Command- und Obersever-Pattern sind bekannte Entwurfsmuster, die bereits von der Viererbande (Gang-of-Four) um Erich Gamma, einer der Hauptfiguren im Umfeld von Eclipse, aufgestellt wurden und in zahlreichen Anwendungen Verwendung finden.

Model-View-Controller (MVC)

Der MVC zählt zu den Architekturmustern und isoliert ein Programm in die drei Einheiten Datenmodell (Model), Präsentation (View) und Programmsteuerung (Controller). Hierdurch soll ein flexibles Programmdesign erreicht werden, um eine spätere Änderung oder Erweiterung einfach zu halten und die Wiederverwendbarkeit der einzelnen Komponenten zu ermöglichen. Bei großen Anwendungen sorgt es für eine gewisse Übersicht und Ordnung

durch Reduzierung der Komplexität. Für kleinere Applikationen bewirkt es leider das Gegenteil. Die Trennung bringt gleichzeitig eine Rollenverteilung mit sich da die Fachkräfte entsprechend ihrer Fähigkeiten eingesetzt werden können.

Command (Action, Transaction)

Das Command Entwurfsmuster gehört zu den Verhaltensmustern. Es kapselt Anfragen als Command-Objekt, um Empfänger hiermit zu parametrisieren. Dies wird oft auch als die objektorientierte Entsprechung zu Callbacks gesehen. Das Erstellen des Befehls und die tatsächliche Ausführung können zu verschiedenen Zeiten oder in verschiedenen Kontexten stattfinden. Auch die Implementierung eines Rückgängig-Mechanismus lässt sich mit dem Command Entwurfsmuster einfach realisieren. Die Command-Objekte werden nach der Ausführung auf einen Command-Stack gelegt und können bei Bedarf eines nach dem anderen Rückgängig gemacht werden. Zu den Vorteilen zählt, dass Auslösender und Ausführender entkoppelt sind. Außerdem können die Befehlsobjekte wie andere Objekte manipuliert und bei Bedarf zu komplexen Befehlen kombiniert werden.

Observer (Dependents, Publish-Subscribe, Listener)

Der Observer ist ein Entwurfsmuster, das zu den Verhaltensmustern zählt und weit verbreitet ist. Es ermöglicht die Weitergabe von Ereignissen an abhängige Objekte. Das observierte Objekt bietet ein Verfahren an, um Beobachter an- und abzumelden und diese über Ereignisse zu informieren. Es kennt seine Beobachter nur über eine Schnittstelle. Es gibt jedes Ereignis völlig unspezifisch an jeden angemeldeten Beobachter weiter ohne dessen genaue Struktur zu kennen. Die Beobachter implementieren ihrerseits eine spezifische Methode, um auf die Änderung zu reagieren. Nützlich ist dieses Entwurfsmuster wenn die Änderung eines Objekts Änderungen an anderen Objekten nach sich ziehen soll. Zu den Vorteilen zählt, dass Beobachtender und Beobachter unabhängig variiert werden können und nur auf eine minimale Arte lose gekoppelt sind. Die Änderungen werden automatisch weitergegeben und es sind auch Broadcasts möglich. Auch können die Abhängigkeiten zwischen Objekten dynamisch zur Laufzeit festgelegt werden. Zu den Nachteilen zählt, dass Änderungen an der Schnittstelle des beobachteten Objekts zu vielen

Änderungen bei den Beobachtern führen. Außerdem kann es zu Endlosschleifen kommen wenn zu viele Benachrichtigungen eintreffen während der Beobachter noch alte abarbeitet. Es kann hierzu kommen wenn zu viele Ereignisse vom beobachteten Objekt ausgelöst werden.

Architektur des GEF

In einem MVC-Design ist der Controller die Verbindung zwischen der View und dem Model. Der Controller ist verantwortlich für die Pflege der View und die Interpretation der Benutzeroberflächen-Ereignisse und deren Umwandlung in Operationen am Model.

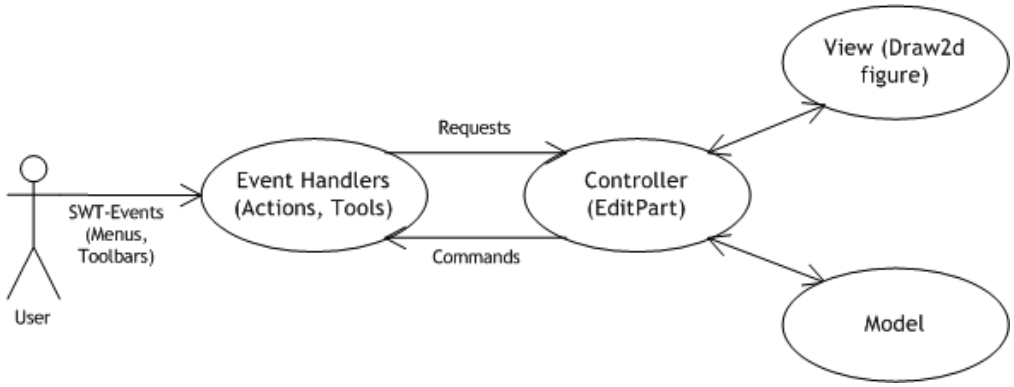

Abbildung 13: Kommunikation im Graphical Editing Framework (GEF)

Model

Das Model repräsentiert die Daten, die persistent gehalten werden sollen (vgl. Majewski, 2004). Das GEF kann fast jedes beliebige Model verwenden. Dieses muss aber einige Anforderungen erfüllen. Es muss über einen Benachrichtigungsmechanismus verfügen, mit dessen Hilfe es den Controller benachrichtigt falls Änderungen am Model aufgetreten sind und die View entsprechend angepasst werden muss. Weiterhin muss ein Mechanismus eingebunden werden, der die Persistenz sicherstellt, wenn der Editor geschlossen wird.

Figures / TreeItems (View)

Mit Hilfe der View werden dem Benutzer die visuell sichtbaren Elemente präsentiert. Diese können vom Anwender aber nicht nur angeschaut, sondern auch modifiziert werden. Als Elemente der View können beim Graphical Editing Framework Draw2d-Figuren oder SWT-

TreeItems verwendet werden. Obwohl die View keine direkten Referenzen zum Model oder Controller besitzen sollte, muss es für jeden relevanten Aspekt des Models, der editierbar sein soll, ein visuelles Attribut haben.

EditParts (Controller)
Beim GEF gibt es für jedes angezeigte Model-Objekt, das editiert werden kann, einen Controller, dessen Name EditPart ist. Er stellt die Verbindung zwischen Model und View dar und ist für das Editieren verantwortlich. Hierzu verfügt er über Helfer mit dem Namen EditPolicies. Die Aufgabe der EditParts ist das Model zu verstehen, Ereignisse über dessen Änderung abzufangen und die View entsprechend zu aktualisieren.

4.2.3 Fazit

Die beiden Architekturen weisen einige Gemeinsamkeiten auf, unterscheiden sich aber auch in vielen Dingen. Das Graphical Editing Framework ist strikter nach dem MVC-Pattern aufgebaut und ist deshalb im Aufbau etwas komplexer. Das Graphics Framework von JViews kapselt Funktionalität und ist nach außen hin einfacher. Dadurch ist es weniger anpassbar als das GEF, dafür aber leichter zu verwenden. JViews ist im Vergleich zum GEF ausgereifter, da es bereits länger existiert.

Beim GEF sind mehr Schritte notwendig um eine Grafik im Editor angezeigt zu bekommen und diese dann editierbar zu machen. Neben der Erstellung der eigentlichen Modell-Klasse muss für diese ein korrespondierender EditPart (Controller) erstellt werden, mit entsprechenden EditPolicies und Commands. Zum Schluss muss noch die eigentliche Figur programmiert werden. Dadurch entsteht zwar mehr Handarbeit, das gesamte System ist aber besser anpassbar. Im Vergleich dazu übernimmt der IlvManager bei JViews viele Aufgaben, ohne dass der Entwickler sich darum kümmern muss. Das IlvManager-Objekt ist allgemein ein sehr zentrales Objekt in der JViews-Architektur.

Das GEF hat viel Zukunftspotential. Die Entwicklung im Rahmen des Eclipse-Projekts wird vorangetrieben und es bauen schon jetzt viele Anwendungen darauf auf. Ein Problem stellt die teils mangelhafte Dokumentation dar (vgl. Bokowski, 2005, S. 40). Viele Dinge werden erst nach einer genaueren Studie des Quellcodes klarer. Dieses Problem hat sich aber in letzter Zeit deutlich gebessert. Eine Verwendung des GEF im Rahmen der Integration erscheint sinnvoll, wird aber mit enormem Aufwand verbunden sein.

5 Evaluierung einzelner Technologien

„Some things are better done than described."

- Andrew Hunt -

Seit Eclipse 3.0 ist es möglich AWT und Swing in Eclipse basierten Anwendungen zu verwenden. SWT (Eclipse) stellt hierzu die Klasse SWT_AWT im Paket org.eclipse.swt.awt zur Verfügung. Diese Klasse dient als Brücke und bietet eine einfache Schnittstelle zwischen SWT- und Swing-Widgets (vgl. Joubert, 2005, S. 18). Sobald erst einmal mit Hilfe eines SWT-Widgets ein Swing-Container angelegt wurde, können alle AWT- und / Swing-Komponenten initialisiert werden. Das ganze funktioniert auch andersherum. Es lassen sich genauso SWT-Widgets in AWT- / Swing-Komponenten einbetten. In beiden Fällen gibt es aber einige signifikante Einschränkungen und Risiken:

- Mit JDK 1.4 funktioniert die Integration nur unter Windows. Für Motif und GTK wird JDK 1.5 benötigt.
- Die Integration funktioniert nicht mit leichtgewichtigen Widgets. Zu diesen zählen beispielsweise die Komponenten für den Zugriff auf OLE (Object Linking and Embedding) oder Toolbars unter Motif.
- Das Paket ist noch im experimentellen Zustand, die API kann sich noch ändern.
- Die Integration ist nicht so eng wie bei der Erstellung der Applikation mit Hilfe des SWT, vor allem was das Look & Feel angeht.

Der Königsweg zur schnellen Integration des Process Modelers in Eclipse bestünde darin, die bereits vorhandenen Software-Assets des Process Modelers so weit wie möglich in Eclipse wiederzuverwenden. Bei den für die Logik zuständigen Klassen sollte es keine nennenswerten Probleme geben. Alle Klassen, die Elemente der Benutzeroberfläche darstellen, müssten aber in SWT umgeschrieben werden, falls eine Integration über die SWT-AWT-Brücke nicht möglich sein sollte. Es muss deshalb evaluiert werden, ob die SWT-

AWT-Brücke alle benötigten Anforderungen erfüllt. Neben der Evaluierung der SWT-AWT-Brücke wird im letzten Teil der Evaluierungsphase auch das Graphical Editing Framework (GEF) genauer analysiert und auf die notwendigen Anforderungen hin untersucht.

5.1 Evaluierung der SWT-AWT-Brücke anhand eines Test-Plug-Ins

Um die bereits vorhandenen Komponenten des Process Modelers übernehmen zu können, muss untersucht werden, ob die SWT-AWT-Brücke einige Anforderungen erfüllt. Es handelt sich hierbei um grundlegende Funktionalitäten, ohne die eine Integration über die SWT-AWT-Brücke nicht realisierbar ist.

Zuerst muss überprüft werden, ob sich Swing-Komponenten wie versprochen in SWT-Komponenten einbetten lassen. Anschließend muss eine weitaus wichtigere Frage geklärt werden: werden Ereignisse wie Maus- und Tastatur-Eregnisse korrekt an die Swing-Komponenten weitergegeben. Ohne diese Funktionalität ist eine sinnvolle Nutzung nicht möglich. Der letzte wichtige Punkt betrifft die Manipulation von Swing-Komponenten aus dem SWT-Rendering-Thread und von SWT-Komponenten aus dem Swing-Rendering-Thread. Da zwei grafische Toolkits verwendet werden, laufen im Hintergrund parallel zwei voneinander getrennte Threads zum Rendern. Es muss möglich sein diese miteinander zu synchronisieren. Der Zugriff von einem Thread auf den anderen muss möglich sein.

Um sich zu überzeugen, ob die SWT-AWT-Brücke die oben genannten Anforderungen erfüllt, wird eine Test-Anwendung entwickelt. Statt eine Dummy-Applikation in SWT zu programmieren, wird gleich ein Test-Plug-In für Eclipse erstellt. Dieses Vorgehen kommt der späteren Umgebung, in der die Integration letztendlich realisiert werden soll, am nähsten. Außerdem können so die ersten Erfahrungen in der Entwicklung von Eclipse Plug-Ins gesammelt werden. Das geplante Plug-In erweitert Eclipse um einen neuen Menüpunkt, einen selbst entwickelten Editor und eine eigene Ansicht. Das Plug-In und der vollständige Quellcode befinden sich auf der CD, die der Diplomarbeit beiliegt.

Implementierung

Innerhalb des Editors und der Ansicht lässt sich relativ einfach eine eigene Oberfläche aufbauen, in diesem Fall eine Swing-Oberfläche eingebettet in ein SWT-Composite. Der Editor erweitert die abstrakte Klasse `EditorPart` und die Ansicht die abstrakte Klasse `ViewPart`. In beiden Fällen steht die Methode `createPartControl(Composite parent)` zur Verfügung, die einfach überschrieben wird um die eigene Oberfläche aufzubauen. Für die Erstellung eines neuen AWT-Frames wird ein `Composite` mit dem Attribut `SWT.EMBEDDED` benötigt. Dieses wird zuerst erstellt. Anschließend wird mit Hilfe der statischen Methode `new_Frame(Composite parent)` der Klasse `SWT_AWT` der benötigte AWT-`Frame` als Child des vorher erzeugten `Composite` erstellt. In diesem `Frame` kann nun in gewohnter Weise eine Swing-Oberfläche aufgebaut werden.

```
public void createPartControl(Composite parent) {
	Composite composite = new Composite(parent, SWT.EMBEDDED);

	java.awt.Frame frame = SWT_AWT.new_frame(composite);

	...
}
```

Listing 4: Initialisierung der SWT-AWT-Brücke

Die Oberfläche des erstellten Editors besteht aus mehreren Labels, zwei Textfeldern, zwei Schaltflächen und einer Canvas-Fläche zum Zeichnen in der Mitte. In der angelegten Ansicht befinden sich zwei Schaltflächen.

Nach Eingabe eines beliebigen Textes in die Textfelder des Editors wird dieser nach betätigen der linken Schaltfläche in die Labels unter der Canvas-Fläche geschrieben. Auf der Canvas-Fläche lässt sich in Paint-Manier zeichnen. Auf diese Weise wurden die Anforderungen „Einbetten von Swing-Komponenten in SWT-Komponenten“ und „Abfangen von Ereignissen durch die Swing-Komponenten“ ausgetestet. Beides funktionierte ohne Probleme.

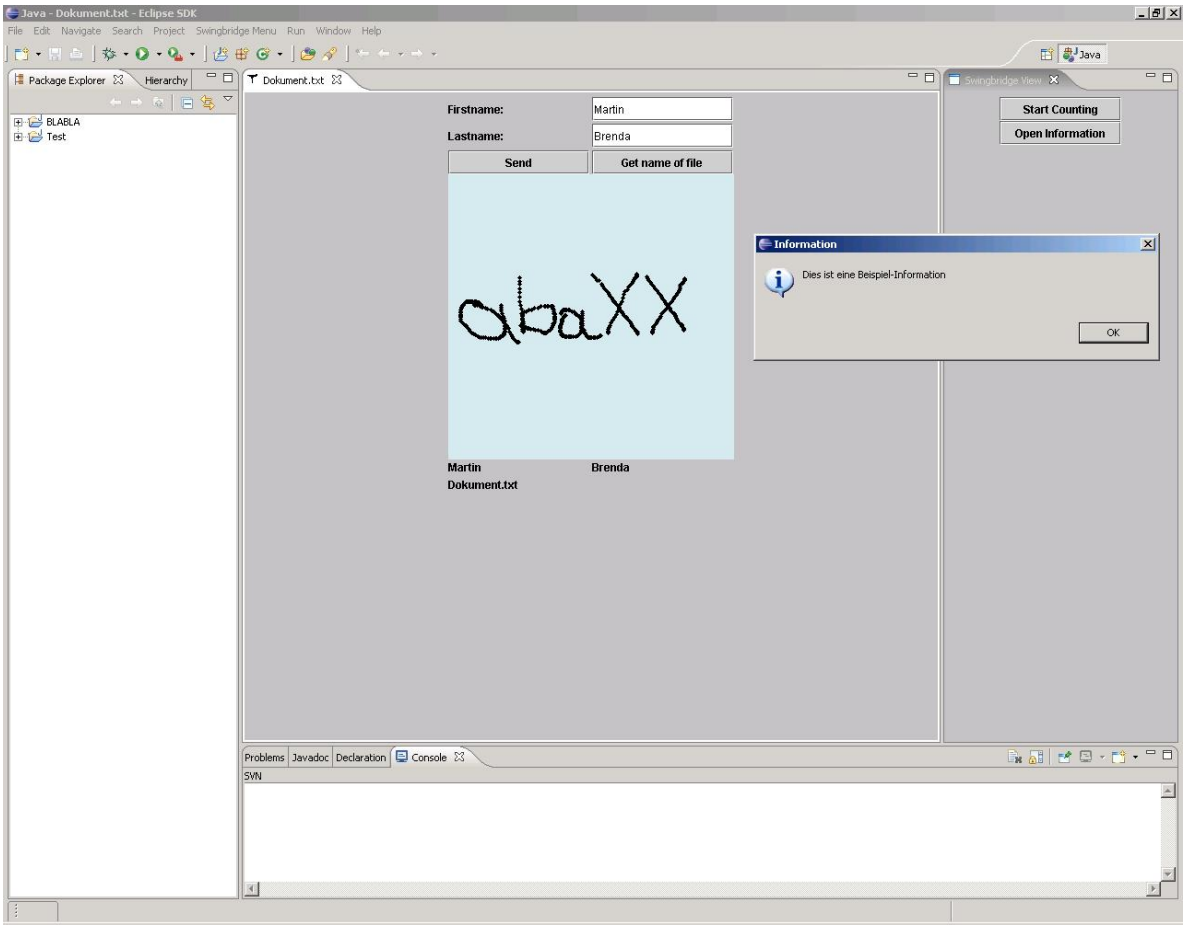

Abbildung 14: Test-Plugin zum Evaluieren der SWT-AWT-Brücke

Wie weiter oben erwähnt wird auch ein zusätzlicher Menüpunkt in das Eclipse-Menü aufgenommen. Nach dem Auswählen dieses Menüpunktes wird ein Text in eines der Swing-Labels des Editors geschrieben. Auf diese Weise wurde die Anforderung „Manipulation der Swing-Komponenten aus dem SWT-Thread“ realisiert. Nach Betätigen der rechten Swing-Schaltfläche im Editor wird der Name der Datei, die mit dem Editor geöffnet wurde, gelesen und ausgegeben. Hierdurch wurde die Anforderung „Manipulation der SWT-Komponenten aus dem Swing-Thread“ realisiert.

Um zu überprüfen ob der „Zugriff auf Elemente des Swing-Rendering-Threads aus einem neu gestarteten Thread“ funktioniert, wird nach Drücken der oberen Schaltfläche in der Ansicht ein neuer Thread gestartet, der einen Text in eines der Label des Editors schreibt. Auch dies funktionierte ohne Probleme. Die zweite Schaltfläche in der Ansicht startet einen neuen Thread, der ein Eclipse-Informationsfenster öffnet und eine Nachricht ausgibt.

Hierzu müssen die beiden Threads synchronisiert werden. Dies geschieht über die von SWT angebotene Methode `Display.getDefault().asyncExec()`. Hierdurch erfolgt die Abarbeitung des gekapselten Codes bei der nächst möglichen Gelegenheit im SWT-Rendering-Thread. Zu beachten ist, dass der aufrufende Thread parallel weiterläuft und nicht von der Beendigung des ausgelagerten Aufrufs benachrichtigt wird. Der Test funktionierte ohne Probleme, so dass die letzte Anforderung „Starten eines neuen Threads aus dem Swing-Code und Zugriff auf Elemente des SWT-Rendering-Thread“ erfolgreich abgeschlossen wurde.

```
class MyThread2 extends Thread {

	public void run() {
		final String titleFrame = ''Informarion'';

		final String message = ''Dies ist eine Beispiel-Information'';

		Display.getDefault().asyncExec(new Runnable() {
			public void run() {


MessageDialog.openInformation(PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell(
), titleFrame, message);
			}
		});
	}

}
```

Listing 5: Synchronisation mit dem SWT-Thread

Die Evaluierung der SWT-AWT-Brücke mit dem einfachen Test-Plugin verlief erfolgreich. Das einzige Problem, das ab und an während der Tests auffiel, war das nicht korrekte Zeichnen der Swing-Oberfläche beim ersten Aufruf des Editors oder der Ansicht. Nach einem Repaint (beispielsweise Fenster minimieren und wieder maximieren) wurde aber alles wieder korrekt dargestellt. Dieses Problem muss im weiteren Verlauf der Integration beobachtet werden.

5.2 Evaluierung der SWT-AWT-Brücke mit JViews-Komponenten

Ein weiterer wichtiger Punkt im Zuge einer Integration über die SWT-AWT-Brücke ist die einwandfreie Funktionsfähigkeit der JViews-Komponenten innerhalb der SWT-Widgets. Nachdem die einfachen Tests abgeschlossen sind kann nun die Untersuchung der JViews-Komponenten in Angriff genommen werden. Es werden im folgenden JViews-Komponenten über die SWT-AWT-Brücke in Eclipse integriert. Die Anforderungen sind die gleichen wie auch schon im vorigen Kapitel.

Es wird wieder ein einfaches Test-Plug-In entwickelt, dieses mal wird aber nur ein Editor zur Verfügung gestellt. Das Plug-In und der vollständige Quellcode konnten der CD nicht beigelegt werden, da für die JViews-Bibliothek eine kommerzielle Lizenz benötigt wird.

Implementierung

Das vorliegende Beispiel baut auf einem Tutorial zur Entwicklung einfacher JViews-Komponenten auf, das sich in der ILOG JViews Dokumentation befindet (vgl. ILOG, 2005, S. 8ff.). Statt eine Swing-Oberfläche aufzubauen, wie im Tutorial beschrieben, wird die Beispiel-Anwendung in einem AWT-Frame realisiert, der in ein SWT-Composite eingebettet ist.

Im ersten Schritt werden ein Manager und eine Ansicht (nicht zu verwechseln mit einer Eclipse-Ansicht, es handelt sich hierbei um eine JViews-Klasse) erstellt. Der Manager verwaltet eine Reihe grafischer Objekte in mehreren Ansichten und Ebenen und ermöglicht Interaktionen (Selektieren, Verschieben, usw.) an den Objekten.

Der erstellte Editor verfügt über die beiden Attribute `ilvManager` (vom Typ `IlvGrapher`) zur Speicherung der grafischen Objekte und `ilvManagerView` (vom Typ `IlvManagerView`) zur visuellen Darstellung des Inhalts des Managers. In der Methode `createPartControl(Composite parent)` wird zuerst der `ilvManager` erstellt und anschließend ilvManagerView. Bei der Erstellung der Ansicht wird dem Konstruktor der

Manager als Argument übergeben. Auf diese Weise wird eine Verknüpfung zwischen Manager und Ansicht erreicht. Zum Schluss werden noch einige optische Werte, wie Antialiasing und die Hintergrundfarbe, modifiziert.

```
public void createPartControl(Composite parent) {
	...

	this.ilvManager = new IlvGrapher();

	this.ilvManager.setInsertionLayer(1);

	this.ilvManagerView = new IlvManagerView(this.ilvManager);

	this.ilvManagerView.setAntialiasing(true);

	this.ilvManagerView.setBackground(Color.white);

	frame.add(new IlvJScrollManagerView(this.ilvManagerView), BorderLayout.CENTER);

	frame.add(this.createButtons(), BorderLayout.SOUTH);

	...
}
```

Listing 6: Erstellung des Managers und der Ansicht

In der Beispielanwendung soll es möglich sein neue grafische Objekte (Rechtecke und Ellipsen) zu erstellen. Auch das Verbinden der Objekte über Kanten soll ermöglicht werden. Hierzu werden drei Schaltflächen in einem `JPanel` unter der Zeichenfläche platziert. Durch das Drücken einer Schaltfläche wird in der Ansicht der gerade aktive Interactor gewechselt. Ein Interactor kann mit einem Werkzeug verglichen werden, zum Beispiel ein Auswahlwerkzeug oder ein Werkzeug zum Erstellen neuer grafischer Objekte. Eine Ansicht hat zu einem bestimmten Zeitpunkt immer einen aktiven Interactor. Er verarbeitet alle Ereignisse innerhalb dieser Ansicht. Nach dem Aktivieren des Interactors zum Erstellen neuer Objekte kann auf der Zeichenfläche das entsprechende Objekt erstellt werden. Listing 7 zeigt die Implementierung der Schaltfläche zur Erzeugung von Rechtecken.

```
public JPanel createButtons() {
	...

	JButton jButtonCreateRectangle = new JButton(''Create Rectangle'');

	jButtonCreateRectangle.setBackground(Color.GRAY);

	jButtonCreateRectangle.addActionListener(new ActionListener() {
		public void actionPerformed(ActionEvent e) {
			if (this.ilvMakeFilledRectangleInteractor == null) {
				this.ilvMakeFilledRectangleInteractor = new
IlvMakeFilledRectangleInteractor();

				this.ilvMakeFilledRectangleInteractor.setGrapherMode(true);
			}

			if (this.ilvManagerView.getInteractor() != this.
ilvMakeFilledRectangleInteractor) {
				this.ilvManagerView.setInteractor(this.
ilvMakeFilledRectangleInteractor);
			}
		}
	});

	...
}
```

Listing 7 - Schaltfläche zum Aktivieren des Interactors zum Hinzufügen eines Rechtecks

Damit die einzelnen Objekte ausgewählt und verschoben werden können wird ein zusätzlicher Interactor benötigt. Hierzu wird eine weitere Schaltfläche in das `JPanel` unter der Zeichenfläche eingefügt. Beim Drücken der Schaltfläche wird der `IlvSelectInteractor` aktiviert und die grafischen Objekte auf der Zeichenfläche lassen sich verschieben und in ihrer Größe verändern. Neben dem Auswählen von Objekten werden zwei zusätzliche Werkzeuge implementiert: `IlvZoomViewInteractor` und `IlvUnZoomViewInteractor`. Mit ihrer Hilfe lässt sich die Zeichenfläche heran- und herauszommen. Die folgende Abbildung zeigt den erstellten Editor mit einem Rechteck und einer Ellipse, die über eine Kante miteinander verbunden sind. Das Auswahlwerkzeug ist gerade aktiv und die Ellipse wurde ausgewählt.

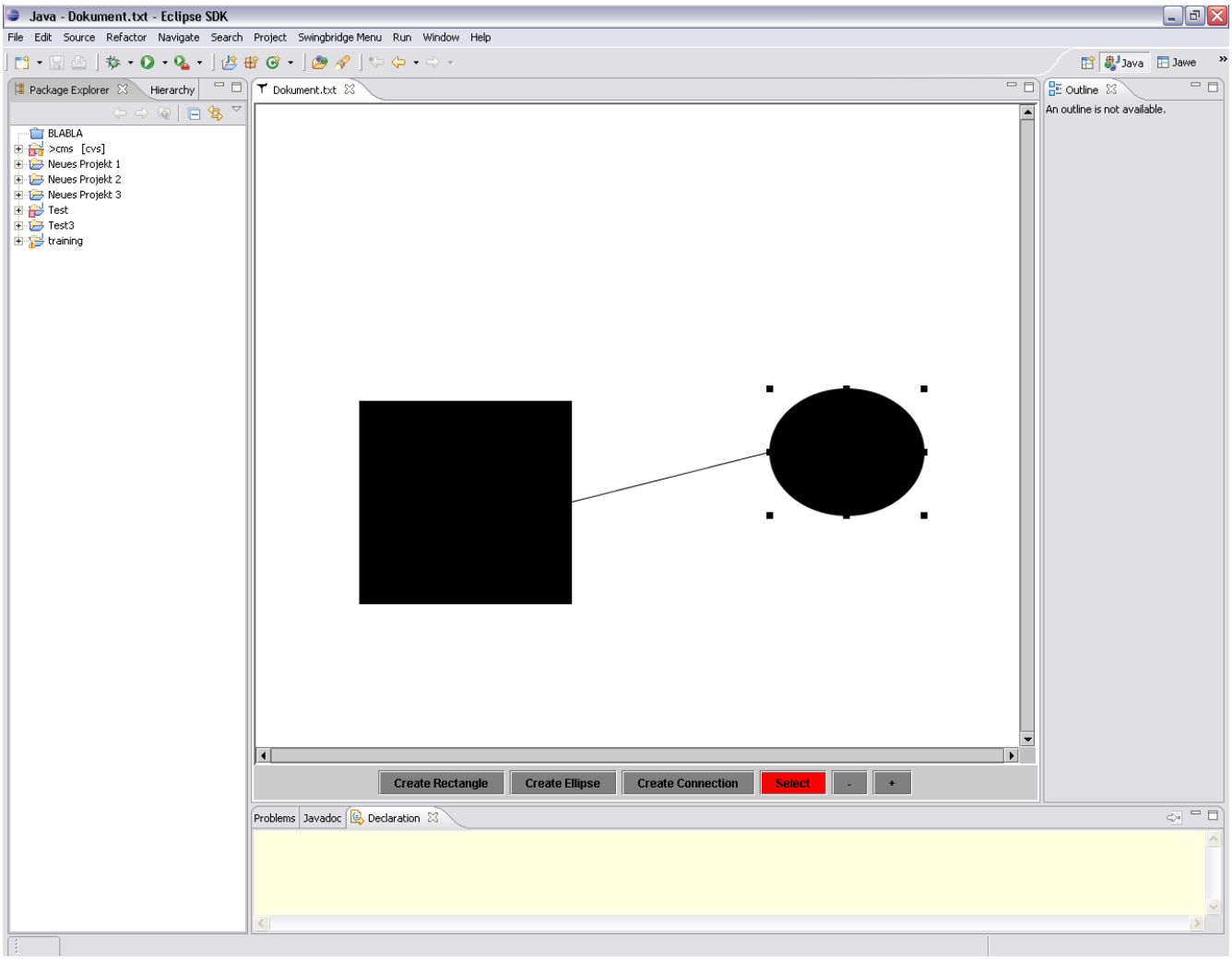

Abbildung 15: Einfacher Editor zum Austesten der JViews-Komponenten

Die Anwendung konnte relativ schnell und einfach erstellt werden, was für ILOG JViews spricht. Vor allem die Dokumentation des Frameworks ist vorbildlich. Alle JViews-Komponenten funktionierten während der Tests ohne Schwierigkeiten. Das einzige Problem war ein leichtes Flackern der grafischen Objekte während des Verschiebens. Auch zahlreiche Versuche mit verschiedenen Buffering-Einstellungen brachten keine nennenswerten Besserungen in dieser Hinsicht. Das Problem liegt wahrscheinlich in der komplexen Struktur von JViews. Einfache Swing-Objekte, wie sie bei der Evaluierung der SWT-AWT-Brücke mit Hilfe des Test-Plug-Ins verwendet wurden, wiesen nicht dieses starke Flackern auf.

5.3 Evaluierung des Graphical Editing Frameworks (GEF)

Das Graphical Editing Framework ist eine Alternative zum derzeitigen grafischen Framework JViews. Eine Evaluierung soll im folgenden klären ob diese Option alle gewünschten Anforderungen erfüllt.

Zu den Anforderungen zählen zuerst einmal grundsätzliche Dinge wie das Zeichnen, Bewegen und Skalieren von grafischen Objekten. Desweiteren muss es möglich sein die grafischen Objekte mit Hilfe von Kanten zu verbinden. Die Ausrichtung der Kanten sollte automatisch erfolgen und diese an anderen Objekten vorbeiführen. Wünschenswert wäre aber auch die Möglichkeit die Kanten von Hand ausrichten zu können, falls die automatische Ausrichtung in komplexen Fällen nicht das gewünschte Ergebnis bringt. Das Framework muss über Werkzeuge wie zum Beispiel ein Auswahlwerkzeug oder Zoom-Werkzeuge verfügen. Ein KO-Kriterium ist auch die Möglichkeit, für ein Modell mehrere Ansichten definieren zu können. Ohne diese Möglichkeit kann keine vernünftige Anwendung erstellt werden, vor allem in Eclipse wo unterschiedliche Ansichten verschiedene Sichten auf das Modell zeigen. Eine weitere, optionale Anforderung, ist die Möglichkeit die grafischen Objekte über XML- oder CSS-Dateien anpassbar zu machen.

Um die oben genannten Anforderungen überprüfen zu können wird ein weiteres Test-Plug-In erstellt. Es bindet einen GEF-Editor mit grundlegender Funktionalität ein. Das Plug-In und der vollständige Quellcode befinden sich auf der CD, die der Diplomarbeit beiliegt.

Implementierung

Der erstellte Editor ähnelt in großen Teilen dem Editor aus der Evaluierung der SWT-AWT-Brücke mit JViews-Komponenten. Er bindet eine Zeichenfläche und eine Toolbar ein. In der Toolbar befinden sich zwei Auswahlwerkzeuge, Schaltflächen zum Hinzufügen von Rechtecken und Ellipsen, sowie eine Schaltfläche zum Erstellen von Verbindungen (Kanten) zwischen den grafischen Objekten. Da die Implementierung des GEF-Editors sehr komplex ist, wird im Folgenden nur auf die allgemeinen Zusammenhänge zwischen Model, View und

Controller im GEF eingegangen und am Schluss das Resultat der Tests zusammengefasst. Eine ausführliche Beschreibung der Evaluierung würde den Umfang der Diplomarbeit unnötig aufblähen.

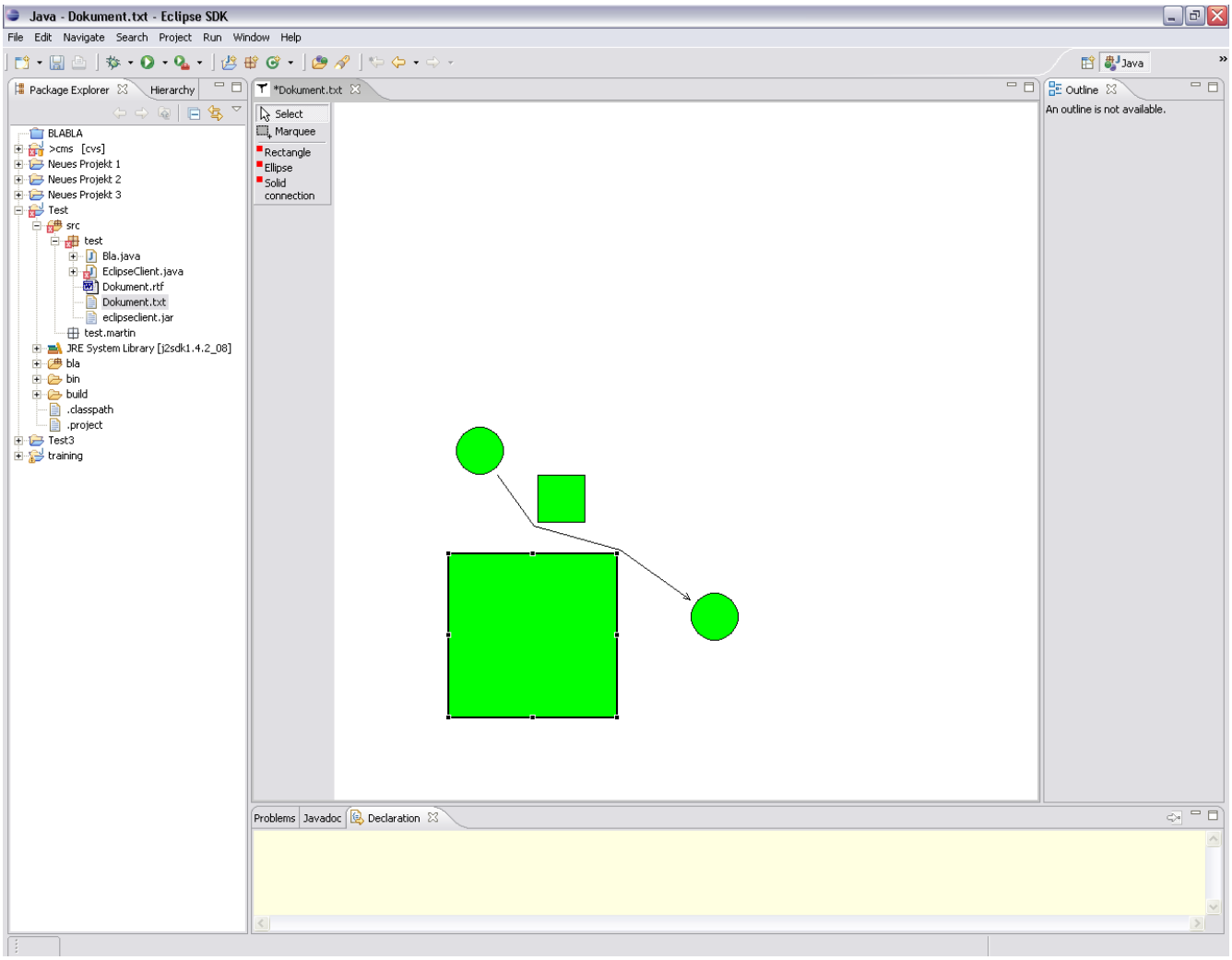

Abbildung 16: Einfacher Editor zum Austesten des GEF

Das Graphical Editing Framework besitzt eine strikte Trennung zwischen Model, View und Controller. Dies fällt schon an den erstellten Klassen auf: für jeden Bereich gibt es eigene Klassen. Das Model kann nach belieben selbst gestaltet werden. Die einzigen Anforderungen sind ein Benachrichtigungsmechanismus, über den der Controller von Änderungen am Model benachrichtigt werden kann, und eine Funktionalität zur Gewährleistung der Persistenz. Der Controller wird über so genannte `EditPart`s realisiert. Jedes Model-Element hat dabei seinen eigenen, ihm zugeordneten `EditPart`. Dieser sorgt für die Verknüpfung zwischen

View und Model. Die View wird beim GEF durch Draw2d-Figuren realisiert. Normalerweise würden hierfür eigene Figuren-Klassen erstellt. Für das vorliegende, einfache Beispiel kann aber auf fertige Figuren von Draw2d zurückgegriffen werden. In diesem Fall Rechtecke und Ellipsen.

Das Model der Beispielanwendung besteht aus den Klassen `Diagram`, `EllipticalShape`, `RectangularShape` und `Connection`, sowie `Shape` und `ModelElement` (siehe Abbildung 17). Die beiden letzten sind abstrakte Klassen die gemeinsame Grundfunktionalitäten aller Model-Elemente zur Verfügung stellen und von denen die konkreten erben.

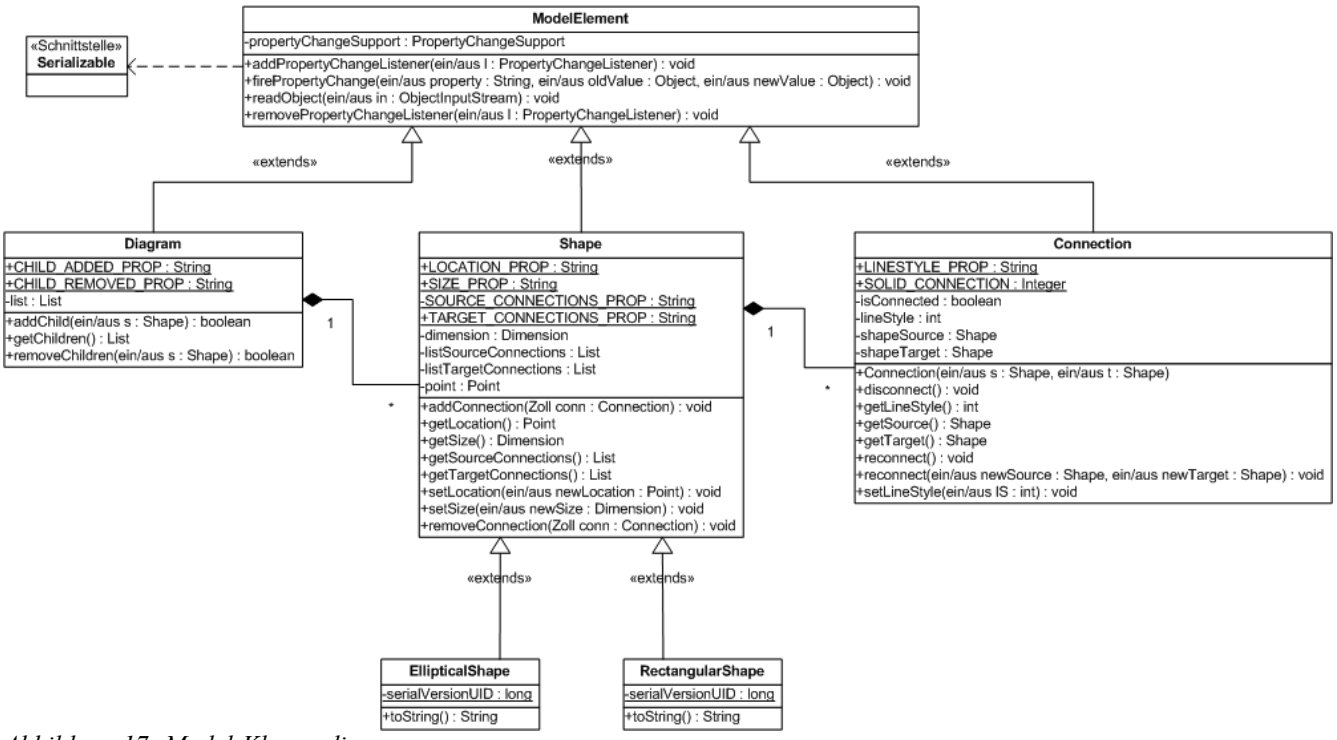

Abbildung 17: Model-Klassendiagramm

Die Klasse `Diagram` stellt den Container dar, in den alle Model-Elemente eingefügt werden. Sie verfügt über eine `ArrayList` zum speichern der Objekte und Methoden zum Hinzufügen und Entfernen von Objekten. Das `Diagram` ist die einzige Instanz, die bei der Erstellung des Editors initialisiert werden muss. Alle übrigen Komponenten werden vom Framework erstellt, sobald die Aufforderung hierfür erfolgt. Die Klasse `ModelElement`,

von der alle erben, stellt den Benachrichtigungsmechanismus zur Verfügung. Dieser wurde mit Hilfe eines `PropertyChangeListener`s auf Basis des Obeserver- (Listener) Pattern realisiert. Die `Shape`s merken sich ihre Position und Größe sowie alle Verbindungen, die zu und von ihnen führen. Die Klasse `Connection` implementiert die eigentliche Verbindung (Kante).

Die Erstellung der `EditPart`s und `Figure`s wird über eine zentrale `EditPartFactory` gesteuert. Bei der Initialisierung der Zeichenfläche (`GraphicalViewer`) im Editor wird dieser eine `EditPartFactory` zugeordnet (siehe Listing 8). Gleichzeitig wird auch das Root-Element des Models an die Zeichenfläche übergeben. Diese erstellt nun für das Root-Element des Models mit Hilfe der `EditPartFactory` einen `EditPart` (Controller), den es dem Root-Element zuordnet. Wenn der Benutzer zum Beispiel ein Rechteck auf die Zeichenfläche zieht wird der gleiche Vorgang gestartet: die Zeichenfläche holt sich über die Factory den entsprechenden `EditPart`.

```
private GraphicalViewer createGraphicalViewer(Composite parent) {
	...

	graphicalViewer.setEditPartFactory(new GEFEditPartFactory());

	graphicalViewer.setContents(this.diagram);

	return graphicalViewer;
}
```

Listing 8: Zuordnung von EditPartFactory und Diagram zur Zeichenfläche

Sobald einem Model-Element ein `EditPart` zugeordnet ist, wird dieses zuständig für die weitere Steuerung. In seine Zuständigkeit fällt die Erstellung und Verwaltung eines grafischen Objekts, die Erstellung und Verwaltung von Kind-EditParts, die Erstellung und Verwaltung von Verbindungen und, wie der Name bereits impliziert, die Unterstützung beim Editieren des Models. So kann sich die Zeichenfläche zum Beispiel über die Methode `createFigure()` eines `EditPart`s ein neues grafisches Element für die View erstellen lassen (siehe Listing 9).

```
protected IFigure createFigure() {
	IFigure f = null;

	if (this.getModel() instanceof RectangularShape) {
		f = new RectangleFigure();
	} else if (this.getModel() instanceof EllipticalShape) {
		f = new Ellipse();
	} else {
		throw new IllegalArgumentException();
	}

	f.setOpaque(true);

	f.setBackgroundColor(ColorConstants.green);

	return f;
}
```

Listing 9: Erstellung einer neuen Figure im ShapeEditPart

Weitere Ausführungen zur Implementierung des GEF-Editors würden den Rahmen der Diplomarbeit sprengen. Zusammenfassen lässt sich sagen, dass das GEF alle am Anfang genannten Anforderungen erfüllt. Nur die optionale Anforderung der Anpassung von grafischen Elementen mit Hilfe von XML- oder CSS-Dateien wird vom GEF nicht unterstützt. Diese Anforderung ließe sich aber bei der Entwicklung eigener Draw2d-Figuren selbst implementieren. Der Aufwand hierfür dürfte nicht allzu groß sein.

5.4 Fazit

Die SWT-AWT-Brücke erfüllt alle notwendigen Voraussetzungen und kann im Rahmen der Integration verwendet werden. Der größte Nachteil der SWT-AWT-Brücke sind die beiden in parallelen Threads ablaufenden Event-Queues (vgl. Joubert, 2005, S. 18). Dies geht soweit, dass sogar einfache Klicks mit Vorsicht gehandhabt werden müssen.

In gewisser Weise zwingt Eclipse die Verwendung von SWT auf, trotz des Vorhandenseins einer SWT-AWT-Brüche. Auf längere Sicht gesehen wird es keinen Weg vorbei an SWT, da die Verwendung mehrerer Toolkits in einer Anwendung nicht sinnvoll ist.

Das Graphical Editing Framework stellt einen vollwertigen Ersatz für ILOG JViews dar. Durch seine schnelle Weiterentwicklung im Rahmen des Eclipse-Projekts bietet es viel potenzial in der Zukunft. Der Nachteil liegt im Arbeitsaufwand für das Ersetzen von JViews durch GEF. Die beiden Frameworks haben leicht unterschiedliche Architekturen. Außerdem sind der derzeitige Editor und das Modell sehr eng mit JViews verknüpft.

6 Integrationsmöglichkeiten

„Sicherheit beruht auf der vermeintlichen Kenntnis und der tatsächlichen Unkenntnis der Zukunft."
- Helmut Nahr -

6.1 Integrationsstufen

Für die Integration einer bereits vorhandenen Anwendung in Eclipse stehen mehrere Möglichkeiten zur Auswahl. Die letztendlich zu treffende Entscheidung, welche Lösung verwirklicht wird, hängt von verschiedenen Faktoren ab. Es spielen die gewünschte Investitionshöhe, Time-to-Market Überlegungen aber auch spezifische Eigenschaften der eigenen Anwendung eine Rolle. Ferner ist das am Ende des gesamten Prozesses gewünschte Ziel von entscheidender Bedeutung.

Eine vollständige Integration vermittelt dem Benutzer das Gefühl einer einzigen Anwendung. Sie ist aber nicht immer wünschenswert. Amsden führt fünf Integrationsstufen auf die im folgenden kurz erläutert werden (vgl. Amsden, 2001). Jede Stufe beschreibt die Art und Weise wie sich eine Anwendung verhält und was der Benutzer erwarten kann.

6.1.1 Keine Integration

Nicht alle Anwendungen erfordern eine Integration (vgl. Amsden, 2001). Eine Applikation welche über eine gute Benutzeroberfläche verfügt und nicht auf andere Anwendungen zurückgreift oder Ressourcen mit anderen Programmen teilt muss nicht unbedingt integriert werden. Es würden hierdurch nur unnötige Kosten und keine wirklichen Vorteile entstehen. Auf dieser Integrationsstufe werden Eclipse und die vorhandene Anwendung unabhängig voneinander benutzt.

6.1.2 Aufruf der Anwendung in einem separaten Prozess

Bei dieser Integrationsstufe wird die vorhandene Anwendung aus Eclipse heraus gestartet (vgl. Amsden, 2001). Eine Möglichkeit ist die Implementierung von Aktionen in Menü- und / oder Symbolleiste bei deren Auswahl die Anwendung startet. Eine andere die Verknüpfung eines bestimmten Ressourcentyps mit dem eigenen Plug-In, so dass beim Öffnen der Ressource diese in der eigenen Anwendung geöffnet wird. Beide Anwendungen müssen den Ressourcentyp unterstützen damit die Daten korrekt angezeigt werden. Der Weg funktioniert natürlich auch anders herum. So kann zum Beispiel aus der eigenen Anwendung heraus der Quellcode einer Klasse in Eclipse geöffnet werden. Der Start der eigenen Anwendung erfolgt in einem separaten Fenster und diese ist ab dem Zeitpunkt des Aufrufs für die Verwaltung der Ressource verantwortlich. Die beiden Anwendungen laufen in unterschiedlichen Java Virtual Machines (JVM) und können deshalb nicht gegenseitig auf ihre Objekte zugreifen. Diese Art der Integration bietet eine einfache Lösung und kann in vielen Fällen als erster Schritt in Angriff genommen werden. Da die Ressource in einem Eclipse-Projekt existiert profitiert die eigene Anwendung von den Vorteilen der Eclipse-Ressourcenverwaltung, wie zum Beispiel Versionierung oder Team-Support.

6.1.3 Datenteilung / Gemeinsames Datenformat

Datenteilung erlaubt die Bearbeitung von Daten durch mehrere Anwendungen. Beim Aufruf der eigenen Anwendung in einem separaten Prozess wurde davon ausgegangen, dass beide Applikationen den vorhandenen Ressourcentyp verstehen. Falls ein proprietäreres Dateiformat verwendet wird, müssen die Voraussetzungen für einen reibungslosen Austausch erfüllt werden. Dies kann durch den Export der Daten in ein von Eclipse unterstütztes Format wie zum Beispiel XML erfolgen oder durch die Erweiterung von Eclipse zur Unterstützung des eigenen Datenformats. Auf diese Weise kann eine Ressource aus beiden Anwendungen heraus bearbeitet werden, ohne dass die beiden Applikationen in irgendeiner Weise miteinander verknüpft sein müssen (vgl. Amsden, 2001).

6.1.4 Integration über gemeinsame API

Bei der API-Integration liegt das Ziel in der Wiederverwertung bereits vorhandener Klassen aus anderen Anwendungen in Eclipse. Klassen die zum Beispiel die Verarbeitungslogik von bestimmten Datenstrukturen zur Verfügung stellen und nichts mit der Benutzeroberfläche zu tun haben können ohne größeren zusätzlichen Aufwand in beiden Anwendungen verwendet werden. Hierzu werden Schnittstellen definiert über die die Klassen angesprochen werden können. Für den Einsatz in Eclipse werden diese Klassen in eine Jar-Datei gepackt und in einem simplen Plug-In gekapselt. Anschließend kann dieses Plug-In in Eclipse von anderen Plug-Ins genutzt werden (vgl. Amsden, 2001). Da die Klassen nun vom Eclipse Plug-In Classloader geladen werden laufen sie in der gleichen JVM wie Eclipse und können ohne Probleme auf die dortigen Daten zugreifen. Auch bei der Entwicklung von Plug-Ins die nur für Eclipse bestimmt sind, ist es sinnvoll das Modell, die Logik und die Benutzeroberfläche in seperate Plug-Ins aufzuteilen. Auf diese Weise können einzelne Teile in Zukunft auch in anderen Kontexten genutzt werden.

6.1.5 Integration in die Benutzeroberfläche von Eclipse

Bei dieser Form der Integration wird die eigene Anwendung komplett in Eclipse integriert, als ob sie ein Teil von Eclipse wäre. Sie kann andere Werkzeuge mitbenutzen und auf alle in Eclipse zur Verfügung stehenden Ressourcen zugreifen. Neue Funktionen können in das Menü und in die Symbolleiste von Eclipse eingebaut werden. Eigene Ressourcentypen werden mit Hilfe von selbst implementierten Editoren aufgerufen und starten im Editorbereich von Eclipse. Zusätzlich kann die eigene Dokumentation in die Eclipse-Dokumentation integriert werden (vgl. Shavor, 2004, S. 215). Für den Benutzer ist es nicht mehr ersichtlich, wo eine Anwendung endet und die nächste beginnt (vgl. Amsden, 2001). Eine vollständige Integration ist umfangreich und kostet einiges an Zeit und Geld. Auch ist es nicht immer einfach die Architektur der eigenen Anwendung eins zu eins in Eclipse zu übernehmen. In einzelnen Fällen kann es zu größeren Änderungen kommen.

6.1.6 Fazit

In den seltensten Fällen passt eine Integration genau in eines der oben beschriebenen Szenarios, oft ist es eine Kombination von mehreren. Deshalb sollen die oben beschriebenen Möglichkeiten nur als Denkanstoß mitgenommen werden. In den meisten Fällen muss eine spezielle, an die eigene Anwendung angepasste, Lösung gefunden werden.

6.2 Beispiele für Integrationen

6.2.1 Poseidon For UML PE

Poseidon For UML PE ist eine Anwendung zum Modellieren von UML-Diagrammen, mit dem getrennte Entwicklergruppen gleichzeitig und ortsunabhängig an demselben Modell eines Softwareprojektes arbeiten und in Echtzeit die Änderungen der Kollegen sehen können. Poseidon For UML ist eine kommerzielle Applikation, die aus der Open Source-Software ArgoUML hervorgeht und diese um etliche Leistungsmerkmale erweitert. Seit der 3er-Version wird auch eine Integration in Eclipse angeboten, wobei diese zusammen mit der eigenständigen Swing-Applikation ausgeliefert wird.

Die Anwendung bietet ein gutes Beispiel für die Integration in Eclipse über die SWT-AWT-Brücke. Denn die meisten Komponenten sind übernommene Elemente aus der Swing-Anwendung. Dies lässt sich schnell an den eingebetteten Swing-Frames mit den nicht zu Eclipse passenden Scrollbalken erkennen. Bevor die Eclipse-Integration verwendet werden kann, muss über die Swing-Anwendung eine Verbindung mit einer Eclipse-Installation hergestellt werden. Hierdurch wird innerhalb der Eclipse-Installation ein Link zu dem Verzeichnis mit dem Poseidon Plug-In erstellt. Dies ist eine Möglichkeit wie Eclipse um eine zusätzliche Install Location erweitert werden kann. Das Poseidon Plug-In befindet sich in einem Unterverzeichnis der Swing-Anwendung. Es muss dort belassen werden und darf nicht in das Plug-In Verzeichnis einer Eclipse-Installation kopiert werden, da es von einigen Bibliotheken der Swing-Anwendung abhängt und die Pfade innerhalb des Plug-Ins relativ

angegeben sind. Nach dem Start von Eclipse ist in der UML-Perspektive erst einmal nichts zu sehen. Um die Eclipse-Integration zu verwenden, muss zuerst über den Navigator oder Package Explorer das UML Modell eines Projekts geöffnet werden. Hierdurch startet Poseidon UML und die Editoren und Ansichten in der UML-Perspektive werden verfügbar. Poseidon lägt im Projektverzeichnis einen Ordner namens uml an, in dem es alle seine Dateien speichert.

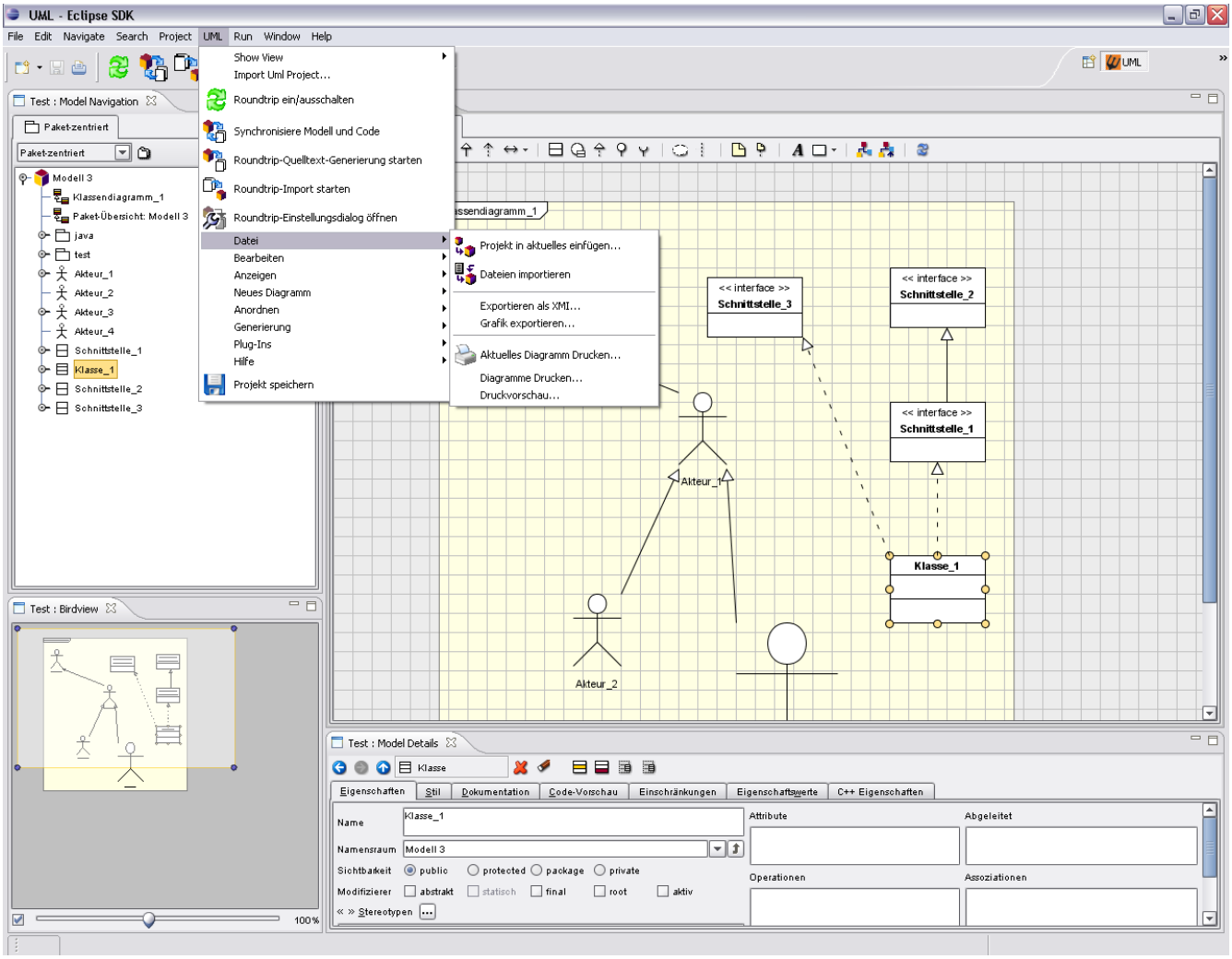

Abbildung 18: Poseidon For UML PE

Der Aufbau der UML-Perspektive ist im groben dem Aufbau der Benutzeroberfläche vom Process Modeler ähnlich. Es gibt eine Zeichenfläche zum Modellieren, eine kleine Übersichtskarte und eine Ansicht mit den Modell-Details. Die Integration scheint auf den ersten Blick gut zu sein, eine genauere Analyse zeigt jedoch einige Schwächen. So sind zum

Beispiel die Icons im Menü und in der Toolbar viel zu groß und passen nicht in das Look & Feel von Eclipse. Auch die Art und Weise wie das Menü der Swing-Anwendung in Eclipse integriert wurde lässt zu wünschen übrig. Es wurde einfach ein neuer Menüpunkt UML hinzufügt und in dieses Menü wurde das gesamte Menü der Swing-Anwendung eingepflanzt, so dass zum Beispiel Menüpunkte wie Ausschneiden und Kopieren doppelt vorkommen: einmal im Edit-Menü von Eclipse und einmal im von Poseidon hinzugefügten Menü UML -> Bearbeiten. Auch die Editoren und Ansichten wurden komplett aus der Swing-Anwendung übernommen und in eingebettete Frames innerhalb von Eclipse-Ansichten eingesetzt. Auf diese Weise sind alle Poseidon-Zeichenflächen in einer Eclipse-Ansicht eingebaut. In dieser wird dann über Tabs navigiert, die von Poseidon stammen. Poseidons Editor zum Modellieren ist also nicht einmal in einen Eclipse-Editor, sondern in eine Ansicht integriert worden. Auch die Einstellungen von Poseidon werden nicht innerhalb des Preferences-Fenster von Eclipse dargestellt, sondern nach dem Drücken eines Schaltknopfs in einem separaten Swing-Fenster gestartet.

6.2.2 Borland Together for Eclipse

Borland Together for Eclipse ist ein Modellierungswerkzeug für alle gängigen UML-Diagrammarten, inklusive Code-Generierung, Reverse- und Roundtrip-Engineering sowie Teamunterstützung. Neben einer in Eclipse integrierten Version gibt es inzwischen auch eine für Visual Studio.

Bei Together wurden die Produkt-Branding-Möglichkeiten von Eclipse voll ausgeschöpft. Auf den ersten Blick sieht die Verzeichnisstruktur aus wie bei einer eigenständigen Anwendung, inklusive mitgeliefertem JRE und eigener Startdatei. Im Hauptverzeichnis befindet sich auch das eclipse-Verzeichnis mit allen Features und Plug-Ins. Nach dem Ausführen der Startdatei wird unter anderem die Lizenz überprüft, bevor der Startvorgang von Eclipse beginnt. Die Integration des eigentlichen Tools in Eclipse ist gut gelungen. Die Benutzeroberfläche ist sehr übersichtlich gestaltet und fügt sich nahtlos in das GUI von Eclipse ein. Nach kurzer Zeit fällt auf, dass Together an sehr vielen Stellen von Eclipse

Erweiterungen deklariert und auch zahlreiche Komponenten von Eclipse mitverwendet. Die Anwendung wurde in etliche Features und Plug-Ins aufgeteilt, fast schon zu viele um noch den Durchblick zu behalten.

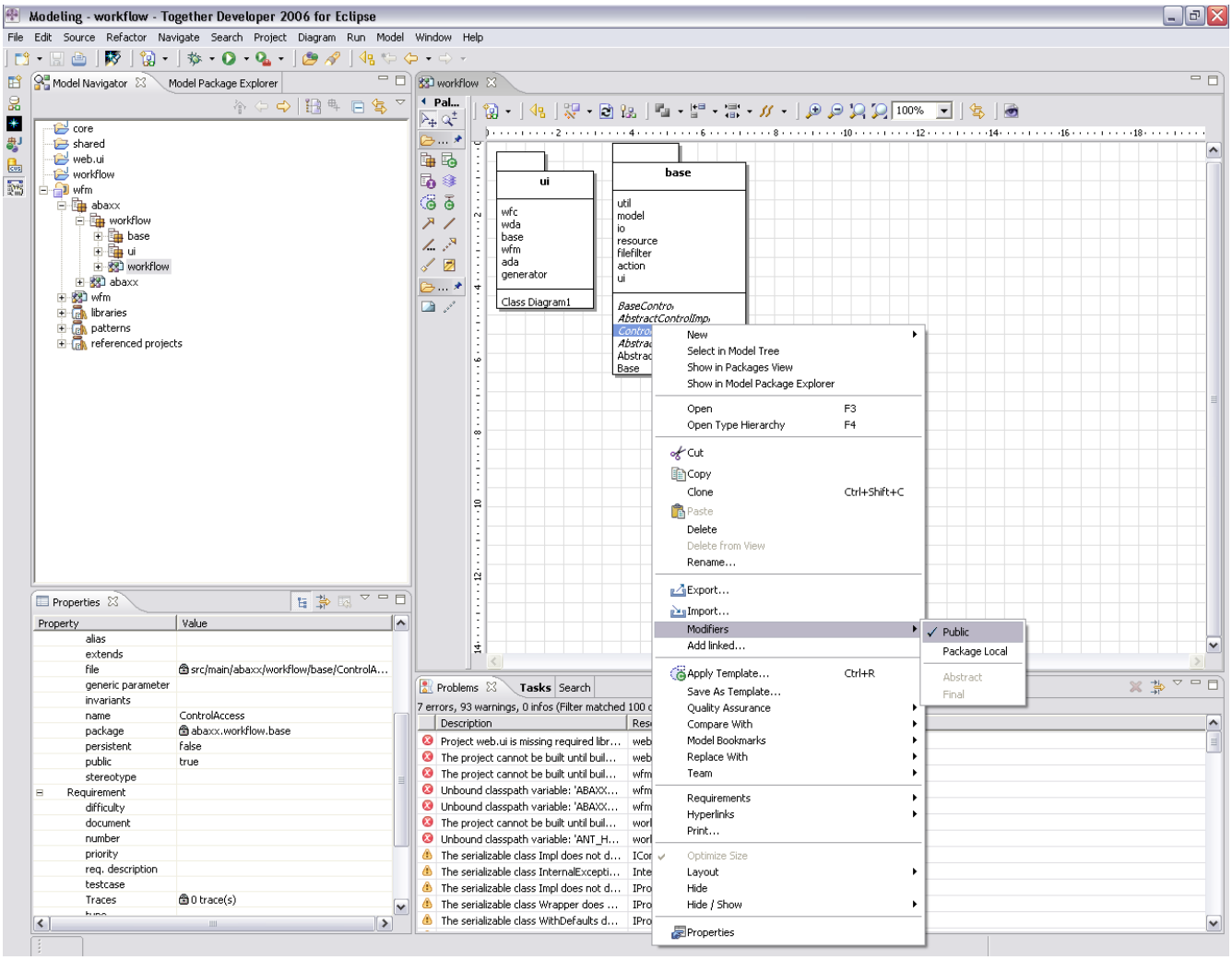

Abbildung 19: Borland Together for Eclipse

Die Anwendung ist zeitweise recht langsam. Einige Anwender klagen zusätzlich, dass nach einer gewissen Laufzeit die gesamte Benutzeroberfläche immer zäher wird. Dies könnte mit dem Lazy-Loading von Eclipse zusammenhängen. Nach einer gewissen Zeit laufen im Hintergrund sehr viele Plug-Ins, was bei leistungsschwächeren Rechnern spürbar wird. Ansonsten lässt sich wenig negatives über die Integration sagen, sie ist insgesamt sehr gelungen.

6.2.3 Fazit

Die beiden untersuchten Integrationen lassen sich klar in zwei Lager unterteilen: Together als sehr gute und gelungene Integration und Poseidon als schnelle, unsaubere Integration einer vorhandenen Swing-Anwendung. Poseidon befindet sich noch relativ am Anfang der Integrationsbestrebungen. Es ist wahrscheinlich nur ein erster Schritt zu einer besseren, vollständigen Integration. Dies zeigt wiederum, das ein Integrationsszenario aus mehreren Schritten bestehen kann. Poseidon zeigt außerdem, dass eine Integration über die SWT-AWT-Brücke sehr wohl machbar ist.

Für den Process Modeler wird der Mittelweg zwischen einer perfekten Integration wie Together und einer sehr schnellen Integration wie Poseidon die beste Lösung sein. Das endgültige Ziel ist aber eine saubere, vollständige Integration wie die von Together. Diese aber in einem Schritt erreichen zu wollen wäre kein guter Ansatz, da sie sehr viel Zeit in Anspruch nehmen und ein gewisses Risiko darstellen würde. Das Ziel ist deshalb ein Integrationsszenario in mehreren Schritten.

6.3 Integrationsszenario für den Process Modeler

Im Fall des abaXX Process Modelers ist das gewünschte Ziel eine vollständige Integration der Anwendung in Eclipse und die Ablösung der derzeitigen Applikation, so dass nicht zwei Produkte nebeneinander existieren und weiterentwickelt werden müssen. Ein weiterer wichtiger Punkt ist der Austausch des grafischen Frameworks, zur Zeit ILOG JViews. Da es sich hierbei um ein kommerzielles Produkt handelt, verursacht dieses Kosten. Entscheidender jedoch ist, dass durch JViews gewisse Grenzen in der Entwicklung auferlegt werden, da gewünschte Funktionalitäten nicht verfügbar sind. Diese können auch nicht selbst implementiert werden, wie zum Beispiel bei einer Open Source-Lösung. Der Austausch des grafischen Frameworks muss nicht sofort erfolgen.

Wegen Time-to-Market Überlegungen kommt eine sofortige, vollständige Integration in Eclipse nicht in Frage. Dies würde einen enormen Zeitaufwand bedeuten und auch ein gewisses Risiko darstellen. Deshalb wird nach einer Lösung gesucht, die sich schrittweise durchführen lässt, so dass am Anfang nur Teile des Process Modelers ausgelagert werden und im späteren Verlauf der Rest nach und nach übernommen werden kann. Auf diese Weise werden der Zeitaufwand und das Risiko minimiert.

Im Folgenden wird ein Integrationsszenario beschrieben, das in vier Schritten eine vollständige Integration des Process Modelers in Eclipse vorsieht. Aus jedem Schritt resultiert ein Zwischenprodukt, das im produktiven Umfeld eingesetzt werden kann. Die einzelnen Schritte bauen aufeinander auf, wobei der zweite und dritte Schritt fließende Übergänge haben. Zu Beginn des gesamten Prozesses werden noch beide Anwendungen parallel nebeneinander weiterentwickelt, der aktuelle Process Modeler und die in Eclipse integrierte Anwendung. Nach dem zweiten Schritt kann die Entwicklung des derzeitigen Process Modelers eingestellt werden, sobald das Plug-In für Eclipse die notwendige Stabilität aufweist. Zusätzlich zu der eigentlichen Integration beschreibt der fünfte Schritt einige Ausbaumöglichkeiten für die Zukunft.

6.3.1 Erster Schritt

Im ersten Schritt werden Eclipse und der Process Modeler locker miteinander gekoppelt. Ziel dieser Lösung ist der Aufruf der Implementierung (Java-Quellcode) einer Aktivität oder der Definition (XML-Datei) eines Parts in Eclipse. Neben der Entwicklung des Plug-Ins sind Änderungen am derzeitigen Process Modeler notwendig. Das Kontextmenü des Process Modelers wird um einen zusätzlichen Menüpunkt *Open in IDE* und weitere Einstellungsmöglichkeiten in den Optionen erweitert. Es handelt sich hierbei um eine einfache und schnell umsetzbare Lösung. Abbildung 20 verdeutlicht die Zusammenhänge zwischen den beiden Applikationen.

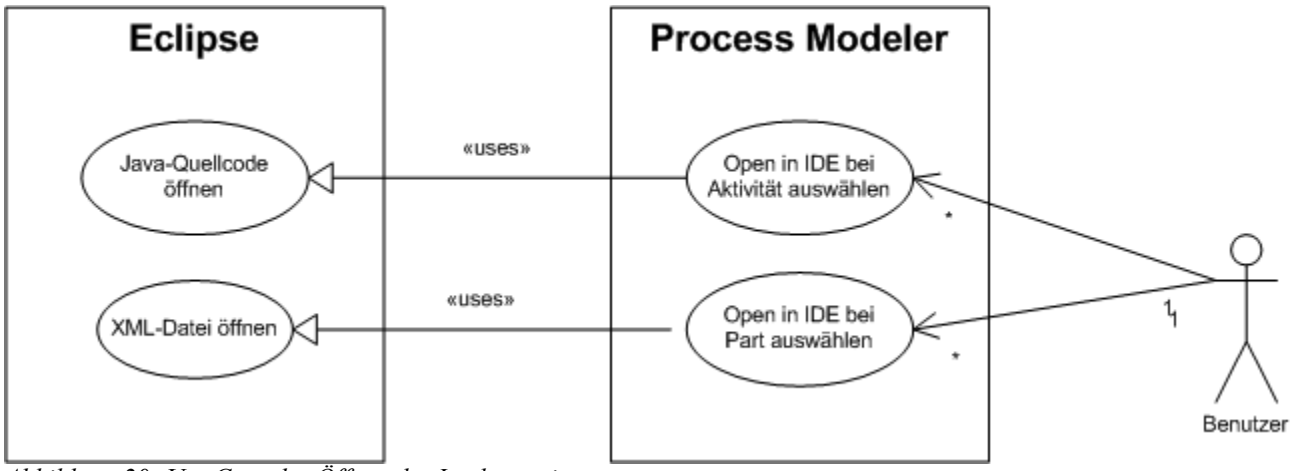

Abbildung 20: Use Case des Öffnen der Implementierung

Die Implementierung soll im richtigen Editor geöffnet werden, zum Beispiel im Java-Editor falls es sich um Java Quellcode handelt oder im XML-Editor wenn es sich um eine XML-Datei handelt. Beim Öffnen der XML-Datei soll direkt in die Zeile gesprungen werden in der der betreffende Part definiert ist. Auf diese Weise erspart sich der Benutzer das Suchen in teils langen Definitionsdateien. Die in diesem Schritt erstellte Lösung kann im späteren Verlauf der Integration wiederverwendet werden und wird somit eine neue Funktionalität darstellen.

6.3.2 Zweiter Schritt

Im zweiten Schritt werden der Editor (ILOG JViews Oberfläche), die Ansichten und die Assistenten des Process Modelers mit Hilfe der SWT-AWT-Brücke in Eclipse übernommen. Auf diese Weise kann das graphische Framework und die meisten anderen Komponenten des Process Modelers weiter verwendet werden, was den Aufwand einer Integration in Eclipse erheblich reduziert. Die Einträge in der Menüleiste und der Symbolleiste von Eclipse müssen durch Eclipse eigene Erweiterungen ersetzt werden. Auch einige der Assistenten, wie zum Beispiel zur Erstellung eines neuen Projekts oder Prozesses, werden vollständig neu erstellt um sich besser in das Eclipse-Framework einzubetten.

Eine Übernahme der einzelnen Komponenten ohne größere Anpassungen wird nicht möglich sein. Die Änderungen fallen aber geringer aus im Vergleich zum Umschreiben aller Komponenten in SWT. Vor allem für die Verwaltung der einzelnen Projekte und die damit zusammenhängenden Probleme wie mehrere Galerien muss ein neues Konzept erstellt werden. Denn im Process Modeler gibt es immer nur ein aktives Projekt, in Eclipse hingegen sind alle offenen Projekte aktiv. Beim Wechsel zwischen den Projekten müssen im Hintergrund die entsprechende Projektkonfiguration aktiviert und die zum Projekt gehörenden Galerien geladen werden. Hierzu wird ein Caching-Mechanismus benötigt, damit der Inhalt der Galerien nicht jedesmal neu aus den XML-Dateien geladen werden muss, in denen er sich befindet. Dies würde die Performance der Anwendung sehr stark verschlechtern.

Der Startvorgang stellt einen weiteren wichtigen Punkt dar. Während des Startvorgangs laufen im Hintergrund Prozesse ab wie die Erstellung des Modells, das Einlesen der Konfiguration oder die Überprüfung des Lizenzschlüssels. Zur Zeit werden alle diese Abläufe beim Startvorgang der Swing-Anwendung durchgeführt. Beim Eclipse-Plug-In wird dieser Vorgang gestartet, sobald das Plug-In das erste Mal aufgerufen wird. Bei der Portierung muss sichergestellt werden, dass alle Vorgänge die im Zusammenhang mit dem Aufbau der Benutzeroberfläche der Swing-Anwendung stehen außen vor gelassen werden und nicht in den Startvorgang des Eclipse Plug-Ins übernommen werden. Aber auch andere Abläufe, die nicht für das Eclipse Plug-In notwendig sind, müssen identifiziert und beiseite gelassen werden.

Die Benutzeroberfläche des Process Modeler Eclipse Plug-Ins soll möglichst an die derzeitige Oberfläche der Swing-Anwendung angepasst werden. Hierzu wird in Eclipse eine eigene Perspektive definiert, die alle Ansichten in gleicher Weise positioniert wie dies in der Swing-Anwendung der Fall ist. Abbildung 21 zeigt einen frühen Entwurf der späteren Perspektive.

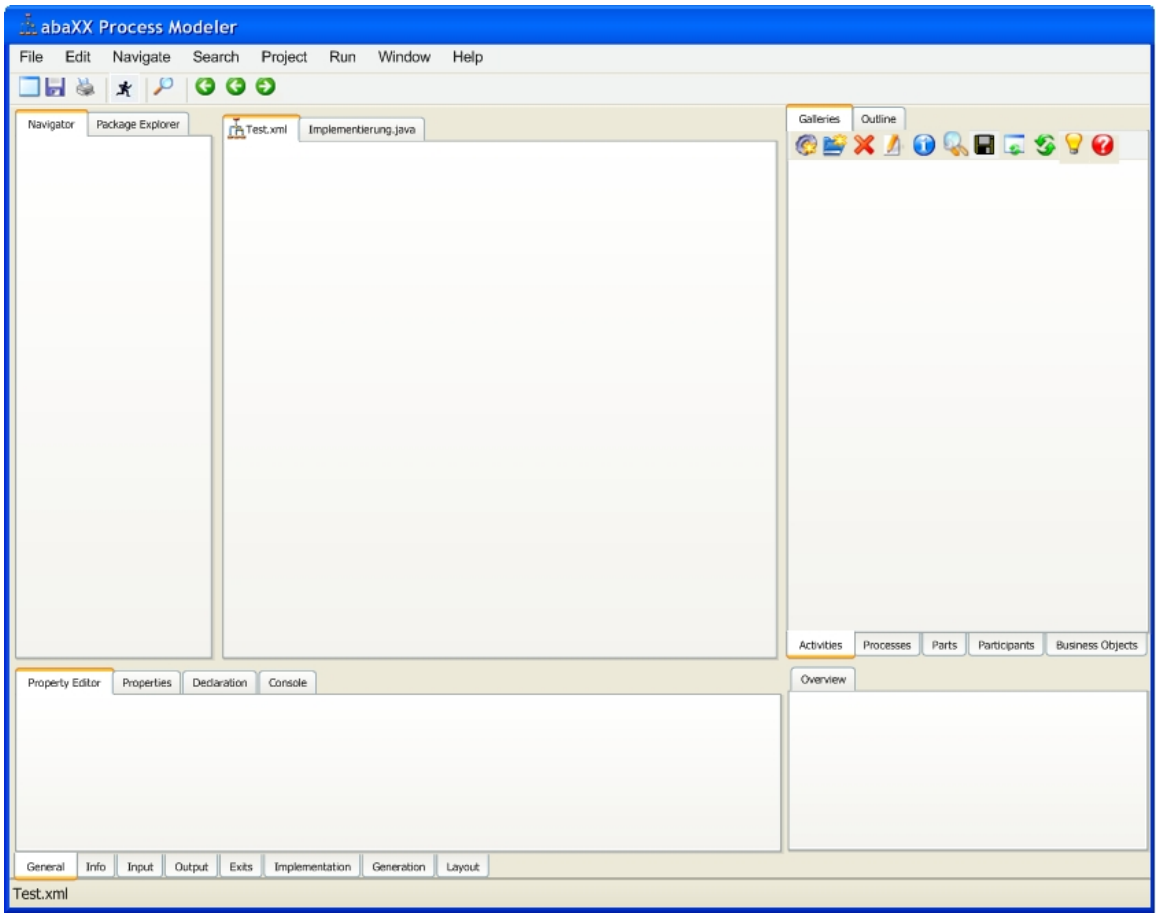

Abbildung 21: Anvisierte Oberfläche für den zweiten Schritt

Um die Auslieferung der neuen Anwendung möglichst einfach zu halten wird ein Process Modeler Feature erstellt. Mit dessen Hilfe lässt sich die Anwendung auf einfache Art und Weise über den Update Manager von Eclipse installieren. Während der Installation werden einige Pfade abgefragt, die das Plug-In zum Laufen benötigt. Ansonsten soll der Benutzer nicht gezwungen sein das Plug-In an einer bestimmten Stelle installieren zu müssen. Er soll es einfach in eine gegebenenfalls bereits vorhandene Eclipse-Installation integrieren können.

Der Aufwand für diesen Schritt ist um einiges größer als der für den ersten Schritt. Gleichzeitig ist er aber um ein vielfaches kleiner zum Ersetzen aller Komponenten durch SWT-Klassen, vor allem da das grafische Framework für die Zeichenfläche vorerst weiter verwendet werden kann und nicht ersetzt werden muss. Bei diesem Schritt wird die Funktionalität aus dem Process Modeler, wie sie bereits existiert, ohne größere

Veränderungen übernommen. Neue Funktionalitäten werden erst im späteren Verlauf des Integrationsszenarios entwickelt. Sobald der zweite Integrationsschritt abgeschlossen ist und das Plug-In eine gewisse Stabilität erreicht, kann auf die eigenständige Swing-Anwendung verzichtet werden.

6.3.3 Dritter Schritt

Im dritten Schritt werden alle Swing-Komponenten, bis auf die Zeichenoberfläche, durch SWT-Komponenten ersetzt. Auf diese Weise wird das Look & Feel verbessert und eine bessere Einbettung in das Eclipse-Framework erreicht. Zusätzlich wird die Hilfe des Process Modelers in das Hilfe-Framework von Eclipse übernommen. In diesem Zusammenhang wird auch eine kontextabhängige Hilfe bereitgestellt.

In diesem Schritt ist außerdem die Entwicklung einer Outline- und Declaration-Ansicht mit Anbindung an die Zeichenoberfläche angedacht. Diese werden in ähnlicher Weise funktionieren wie die Outline- und Declaration-Ansichten des Java Development Toolkits (JDT). Die Outline-Ansicht soll alle Aktivitäten, Prozesse, Parts, Teilnehmer und Business Objekte des gerade offenen Editors in einer gegliederten Form auflisten. Hierdurch wird die Navigation in extrem großen Workflow-Diagrammen erleichtert. Durch Auswahl eines bestimmten Objekts in der Outline wird in der Zeichenoberfläche zum betreffenden Objekt gesprungen und dieses markiert. In der Declaration-Anischt wird die Implementierung eines Objekts angezeigt, sobald ein Objekt auf der Zeichenoberfläche ausgewählt wurde. Eine Änderung der Implementierung ist hier nicht möglich, diese Ansicht dient nur dem schnellen Nachschauen. Zum Editieren der Implementierung wird diese per Doppelklick auf ein Objekt in einem separaten Editor geöffnet. Bei Java-Klassen im Java-Editor des JDT. Falls es sich um ein neu erstelltes Objekt ohne Implementierung handelt soll der Assistent zum Erstellen neuer Java-Klassen gestartet werden. Hierdurch wird Schritt für Schritt die Einbettung in Eclipse und die Zusammenarbeit mit anderen Werkzeugen verbessert.

Auch die Umprogrammierung aller Komponenten in SWT erfordert einen gewissen Zeitaufwand und wird nicht in kürzester Zeit durchführbar sein. Der Vorteil ist, dass bereits ein Plug-In zur Verfügung steht das im produktiven Umfeld verwendet werden kann und die Weiterentwicklung der einzelnen Komponenten nach und nach nebenbei erfolgen kann.

6.3.4 Vierter Schritt

Der letzte Schritt auf dem Weg zu einer vollständigen Integration ist das Auswechseln des graphischen Frameworks (ILOG JViews) durch das Eclipse eigene Graphical Editing Framework (GEF). Hierdurch wird eine bessere Aufteilung des Quellcodes nach dem MVC-Pattern erreicht. Desweiteren fallen die Lizenzkosten für ILOG JViews weg und die Möglichkeiten der Open Source-Welt stehen offen.

Das Auswechseln des Editors zum Zeichnen der Workflows ist eine aufwändige Prozedur die umfangreiche Anpassungen am Quellcode erfordert. Zwar können einzelne Teile übernommen werden, doch wegen der unterschiedlichen Architektur des GEF im Vergleich zu JViews und der strikteren Trennung des Quellcodes nach dem MVC-Pattern müssen viele Teile umgeschrieben werden. Ein weiteres Problem ist die enge Verknüpfung des derzeitigen Modells mit JViews. Nach einer Abspaltung von den JViews-Klassen bleibt nicht mehr viel übrig vom Modell, so dass gegebenenfalls ein vollständig neues Konzept für das Modell erstellt werden muss / kann. Genaueres lässt sich aber erst nach einer umfassenden Analyse des Modells sagen, die nicht Teil dieser Diplomarbeit ist.

6.3.5 Fünfter Schritt

Der fünfte Schritt zählt nicht direkt zur Integration. Hier werden einige Ausbaumöglichkeiten für die zukünftige Entwicklung aufgeführt. Die meisten von ihnen nutzen die von Eclipse bereitgestellten Features und bauen auf ihnen auf.

Eine Überlegung, die bereits jetzt schon angedacht wird, wäre die Trennung der Anwendung in Developer- und Designer-Version, ganz im Sinne eines rollenbasierten Entwicklungsprozesses. Ein ähnliches Prinzip verfolgt beispielsweise Borland Together. Dort gibt es bereits zwei unterschiedliche Anwendungen: eine für Entwickler, deren Interesse mehr in der Implementierung liegt, und eine weitere für die Beschäftigten in höheren Positionen und Analysten, die mehr an den gesamtheitlichen Abläufen und der Modellierung von Anforderungen interessiert sind. Neben der Auftrennung in zwei unterschiedliche Anwendungen ließe sich so etwas auch durch verschiedene Perspektiven in Eclipse erreichen.

Eine weitere, sehr interessante Funktionalität, wäre das Debugging von Prozessen. Vorstellbar wäre eine Perspektive ähnlich der von Java-Debug, mit entsprechendem Funktionsumfang. Hierzu müsste Eclipse mit der abaXX Workflow Engine verknüpft werden, um Prozesse ablaufen lassen zu können. Bei jeder Aktivität könnte ein Breakpoint gesetzt werden, um dort zur Laufzeit die aktuellen Daten im Prozess-Context beobachten zu können. Versuche in diese Richtung wurden bereits mit der Entwicklung einfacher Prototypen unternommen. Zur Zeit gestaltet sich der Einblick in einen ablaufenden Prozess relativ schwierig. Die einzige Möglichkeit besteht im Hinzufügen von `System.out.println()` Anweisungen in den Implementierungsklassen.

Zur Zeit werden dem Entwickler alle Prozesse über die Process Gallery zur Verfügung gestellt. Diese speichert deren Informationen in einer XML-Datei, die sie bei der Initialisierung ausliest. In dieser XML-Datei steht unter anderem auch wo ein Prozess aufd der Festplatte gespeichert ist. Bei Änderungen direkt im Dateisystem stimmen die Informationen natürlich nicht mehr überein. Außerdem darf der Entwickler nicht vergessen die Process Gallery bei Änderungen abzuspeichern. Viel vorteilhafter wäre es, wenn die in einem Projekt vorhandenen Prozesse beim Starten des Plug-Ins indexiert würden. Das Plug-In würde hierzu das Dateisystem auf existierende Prozesse hin durchsuchen und diese in einen Cache laden. Jeder Prozess wäre mit einer Datei auf der Festplatte verknüpft. Beim Löschen des Prozesses würde auch die Datei gelöscht und andersherum. Zur Zeit besteht

das Problem, dass die Prozessdateien die Endung XML besitzen. Folglich könnte ein Indexierer nicht zwischen Prozessen und anderen XML-Dateien unterscheiden. Eine Änderung der Dateiendung ist aber bereits angedacht.

Um das Look & Feel und die Einbettung des integrierten Process Modelers zu verbessern, wäre es sinnvoll einige Icons und Grafiken zu überarbeiten und an die Standards von Eclipse anzupassen. Hierzu wurden von Eclipse User Interface Guidelines definiert, die sich hauptsächlich an Designer und Entwickler von Benutzeroberflächen richten (vgl. Edgar, 2004).

6.3.6 Fazit

Innerhalb der Diplomarbeit wird es kaum möglich sein, eine vollständige Integration in Eclipse durchzuführen. Ziel ist es den ersten Integrationsschritt abzuschließen und einen möglichst großen Teil des zweiten abzuarbeiten.

7 Implementierung

„A program is never less than 90% complete, and never more than 95% complete."
- Terry Baker -

Im folgenden werden die ersten beiden Integrationsschritte aus dem vorigen Kapitel implementiert. Bei der Betrachtung des ersten Schritts wird auch auf die allgemeinen Dinge wie das Anlegen eines neuen Plug-In-Projekts oder die Definition von Erweiterungen unter Eclipse eingegangen. Da der Quellcode sehr umfangreich ist wurde auf eine detaillierte Besprechung jeder einzelnen Klasse und Methode verzichtet. Nur die wichtigsten Punkte werden näher beleuchtet.

7.1 Implementierung des ersten Schritts

Es gibt in Eclipse keine spezielle API, die es erlaubt Daten von externen Anwendungen anzunehmen oder anzufordern. Für die Implementierung des ersten Schritts ist es aber notwendig, dass der Name bzw. Pfad der zu öffnenden Datei vom Process Modeler an Eclipse übergeben wird. Eine Lösung für dieses Problem ist die Implementierung einer lokalen Socket-Schnittstelle. Eclipse fungiert dabei als Server, der Process Modeler als Client. Damit der Endbenutzer den Port dieser Verbindung selbst einstellen kann, wird sowohl in Eclipse als auch im Process Modeler eine entsprechende Einstellungsseite implementiert.

Eine weitere Voraussetzung, die erfüllt sein sollte, aber nicht muss, ist die Existenz der zu öffnenden Datei in einem Projekt unter Eclipse. Ist dies nicht der Fall und handelt es sich zum Beispiel um eine Java-Datei, so kann diese nicht im Java Editor von Eclipse geöffnet werden. Sie wird dann zwar geöffnet, aber in einem einfachen Text Editor ohne Syntax-Highlighting.

Aus dem Process Modeler heraus kann die Implementierung von Aktivitäten, Parts und Business Objekten über das Kontextmenü des jeweiligen Objekts in Eclipse geöffnet werden. Hierzu wurde dem Kontextmenü die neue Aktion OPEN IN IDE hinzugefügt. Der Benutzer des Process Modelers kann einstellen welche IDE er benutzt. Falls er nicht Eclipse benutzt, wird der Eintrag OPEN IN IDE erst gar nicht im Kontextmenü angezeigt. Im folgenden wird nur auf den Eclipse-Teil der Implementierung eingegangen. Das fertige Plug-In sowie der gesamte Quellcode des ersten Integrationsschritts befinden sich auf der beiliegenden CD.

7.1.1 Erstellung eines Plug-In-Projekts

Nach der Installation des Eclipse SDK steht das Plug-In Development Environment (PDE) zur Verfügung. Das PDE ist ein Werkzeug welches Entwicklern die Erstellung, das Testen und die Auslieferung von Eclipse-Plug-Ins erleichtern soll. Es erweitert Eclipse um neue Ansichten, Editoren und Assistenten. Vor allem der Editor für die Manifest-Datei erleichtert die Arbeit ungemein.

Für die Entwicklung von Eclipse-Plug-Ins eignet sich das Plug-In-Projekt am besten. Es kann angelegt werden wenn das PDE Teil der Eclipse-Installation ist. Ein neues Plug-In-Projekt kann in Eclipse über FILE | NEW | PLUG-IN PROJECT erstellt werden. Unter PROJECT NAME muss der Name des Projekts eingegeben werden. Namen von Plug-Ins folgen bei Eclipse der Namenskonventionen von Paketen in Java. Ein korrekter Name wäre zum Beispiel *abaxx.workflow.tools.eclipse.wfmlistener*. Alle übrigen Einstellungen können gelassen werden wie sie sind.

Nach dem Drücken von NEXT wird die nächste Seite des Assistenten aufgerufen. Unter PLUG-IN NAME wird *Process Modeler Integration* und unter PLUG-IN PROVIDER *abaXX Technology AG* eingetragen. Über FINISH wird das neue Projekt im Workspace erzeugt.

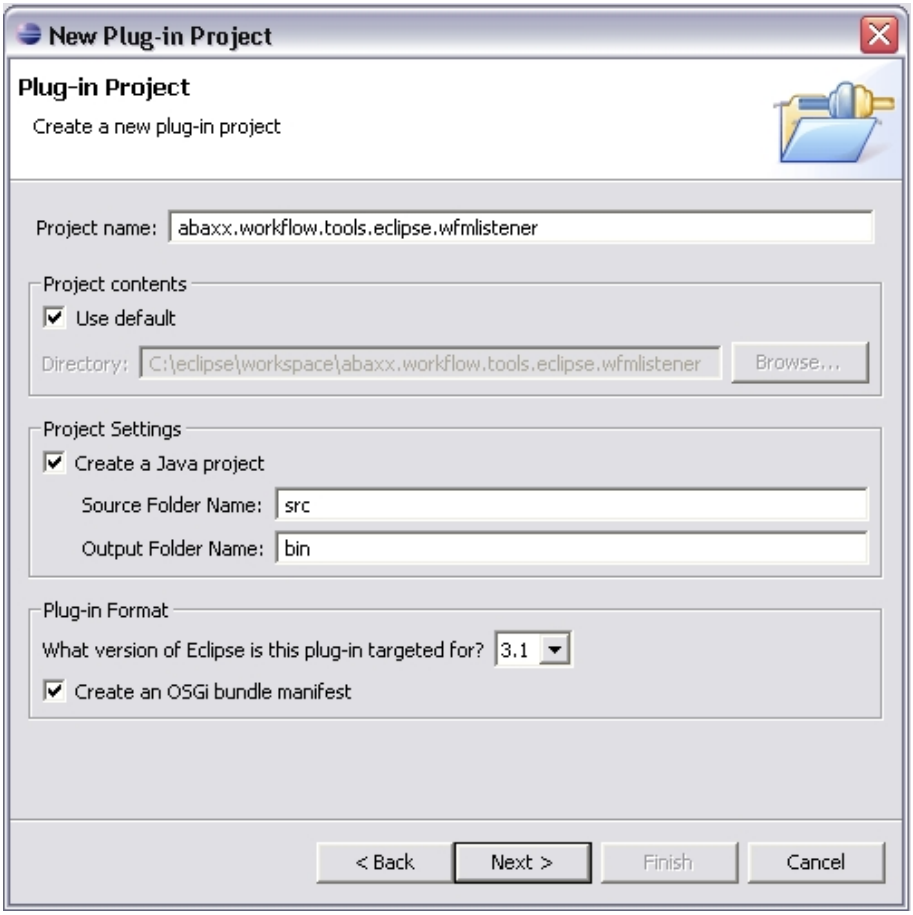

Abbildung 22: Assistent zum Erstellen eines neuen Projekts

Nach dem Anlegen des Projekts existiert bereits die Klasse `WfmlistenerPlugin` (siehe Abbildung 23). Sie erweitert `AbstractUIPlugin` und verwaltet den Lebenszyklus eine Plug-Ins. `AbstractUIPlugin` stellt Strukturen zur Verwendung von UI-Ressourcen bereit. Beim Initialisieren des Plug-Ins wird die Methode `startup()` aufgerufen, beim Beenden `shutdown()`. In diesen Methoden können alle Abläufe implementiert werden, die später global verfügbar sein müssen, beispielsweise das Einlesen von Konfigurationsdateien oder die Überprüfung von Lizenzschlüsseln. Für die vorliegende Aufgabenstellung wird die Klasse `WfmlistenerPlugin` Schritt für Schritt mit weiterer Funktionalität ausgebaut und durch zusätzliche Klassen ergänzt. Eine weitere Datei, die bereits nach dem Anlegen des Projekts existiert, ist *MANIFEST.MF*. Sie beinhaltet die Meta-Informationen des Plug-Ins, wie zum Beispiel den Namen.

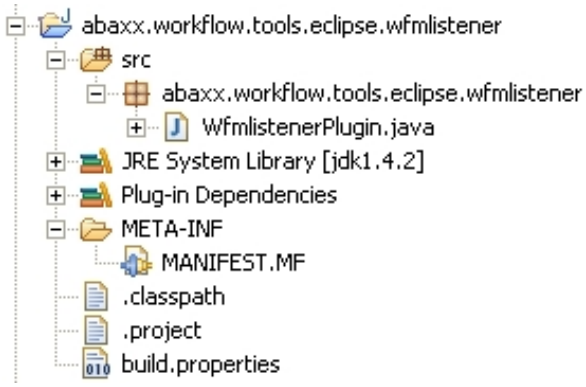

Abbildung 23: Struktur des angelegten Projekts

7.1.2 Der Startvorgang

Normalerweise werden Plug-Ins von Eclipse erst aktiviert, wenn diese gebraucht werden. Dieses Prinzip nennt sich Lazy Loading und verbessert die Performance des gesamten Frameworks. Das Plug-In für die Integration des Process Modelers muss aber schon beim Start von Eclipse aktiviert werden, damit der Port für die Übergabe der Daten vom Process Modeler an Eclipse geöffnet und auf Verbindungen gewartet wird. Hierzu kann ein Erweiterungspunkt von Eclipse verwendet werden.

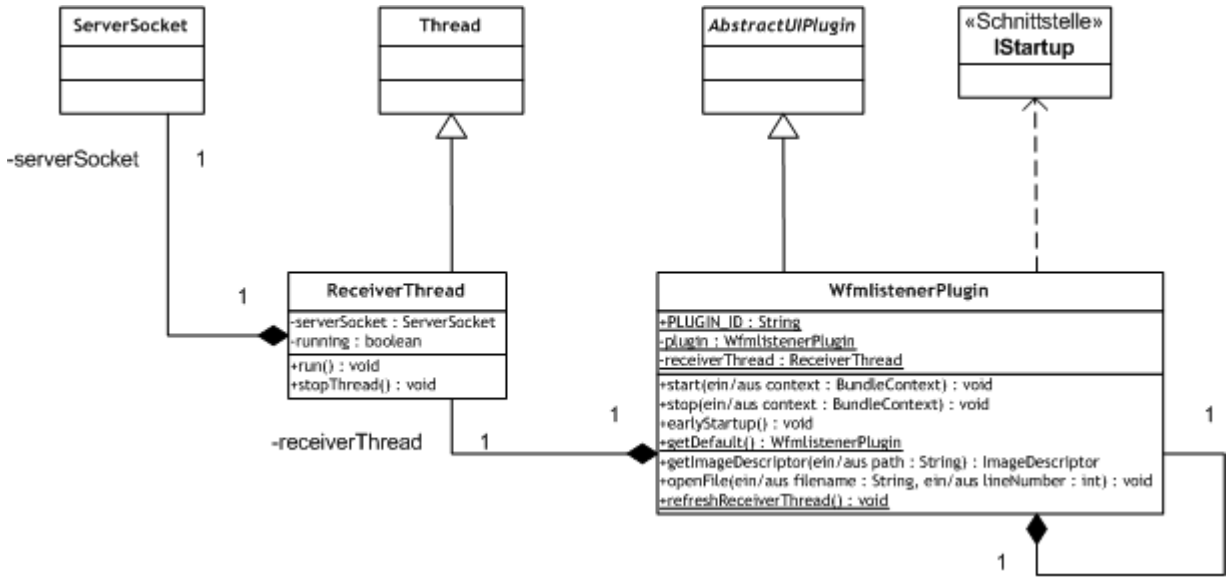

Abbildung 24: WfmlistenerPlugin- und ReceiverThread-Klassendiagramm

Nach dem Öffnen der Manifest-Datei erscheint ein spezieller Editor zum Anpassen der Eigenschaften des Plug-Ins. Über den Karteireiter EXTENSIONS können neue Erweiterungspunkte implementiert werden. Nach Drücken von ADD... und Auswahl des Erweiterungspunkts *org.eclipse.ui.startup* wird dieser über FINISH hinzugefügt. Durch Rechtsklick auf den neuen Eintrag und Auswahl von NEW | STARTUP wird ein neues Element hinzugefügt. An diesem muss nichts geändert werden. Mit Hilfe dieser wenigen Schritte wird dem Framework mitgeteilt, dass das Plug-In bereits beim Start von Eclipse aktiviert werden soll. Neben der Aktivierung des Plug-Ins ruft das Framework die Methode `earlyStartup()` auf und führt die dortigen Aktionen aus. Diese Methode erhält die Klasse `EclipsePlugin` durch die Implementierung der Schnittstelle `IStartup`. In dieser Methode wird ein neuer `ReceiverThread` erstellt und gestartet.

```
public void earlyStartup() {
	try {
		this.receiverThread = new ReceiverThread();

		this.receiverThread.start();
	} catch (IOException e) {
		DebugPlugin.log(e);
	}
}
```

Listing 10: Wird beim Start von Eclipse ausgeführt

Die Klasse `ReceiverThread` ist für das Öffnen des Ports und lauschen an diesem verantwortlich. Den Port bekommt die Klasse aus den Grundeinstellungen von Eclipse. Standardmäßig ist der Port auf 7400 gesetzt, der Benutzer kann ihn aber über die Grundeinstellungen von Eclipse ändern. Hierzu wurde eine Grundeinstellungsseite implementiert auf die im späteren Verlauf näher eingegangen wird. Die Variable `running` wird für das Beenden des Threads benötigt. Sie wird bei jedem Durchgang in der `run()`-Methode überprüft, bei `false` endet die dortige Endlosschleife. Dies ist nötig, da beim Ändern des Ports in den Grundeinstellungen das alte Socket und der alte Thread geschlossen werden müssen und ein neues Socket auf dem nun aktuellen Port geöffnet werden muss. In der `run()`-Methode wird die Verbindung vom Process Modeler angenommen

und der übergebene Befehl wird abgearbeitet. Im Anschluss wird die Methode `openFile()` der Klasse `EclipsePlugin` mit dem Paketnamen inklusive Klassennamen bzw. dem Pfad zum Dokument und der Zeilennummer als Parameter aufgerufen.

```
public ReceiverThread() throws IOException {
	super();

	this.setDaemon(true);

	String port =
EclipsePlugin.getDefault().getPreferenceStore().getString(''listener_port'');

	if (!port.equals('''')) {
		this.serverSocket = new ServerSocket(Integer.parseInt(port));
	} else {
		DebugPlugin.logDebugMessage(''Could not read port from PreferenceStore'');
	}

	this.running = true;
}
```

Listing 11: Öffnen des lokalen Ports

7.1.3 Das Anzeigen von Dokumenten

Um die Abläufe beim Anzeigen von Dokumenten in Eclipse besser zu verstehen folgt ein kurzer Ausflug hinter die Kulissen eines Editors. Das Anzeigen von Dokumenten in Eclipse ist strikt nach dem Model-View-Controller-Pattern getrennt. Das anzuzeigende Dokument repräsentiert das Model, der Editor den Controller und der formatierte, angezeigte Text die View. Die Abbildung zwischen dem Eingabedokument und dem Editor stellt der Dokument-Provider her. Er erzeugt und verwaltet den Dokumentinhalt und benachrichtigt die Editoren über Änderungen am Dokumentmodell. Jedem Dokument-Provider ist ein Eingabetyp zugeordnet den er verarbeiten kann. Dieser Eingabetyp ist eine Abstraktion des anzuzeigenden Dokuments.

Normalerweise lassen sich in Eclipse nur Dateien öffnen, die in einem Eclipse-Projekt existieren. Um beliebige Dateien aus dem lokalen Dateisystem öffnen zu können, wurden die Klassen `External`, `ExternalEditorInput` und `ExternalDocumentProvider` implementiert. Die Klasse `External` erbt von `PlatformObject` und implementiert die Schnittstelle `IStorage`. Sie repräsentiert eine externe Datei vergleichbar mit `IFile`, die aber im Gegensatz dazu eine Datei in Eclipse repräsentiert. Die beiden Methoden `getContents()` und `setContents()` sind für das Auslesen und Schreiben der Datei verantwortlich. Die Klasse `ExternalEditorInput` ist der vom Dokument-Provider benötigte Eingabetyp. Sie erbt von `IStorageEditorInput` und kapselt das anzuzeigende Dokument vom Typ `External`. Die Klasse `ExternalDocumentProvider` ist der Dokument-Provider selbst. Die Methode `doSaveDocument()` wird überschrieben, um das externe Dokument richtig zu speichern.

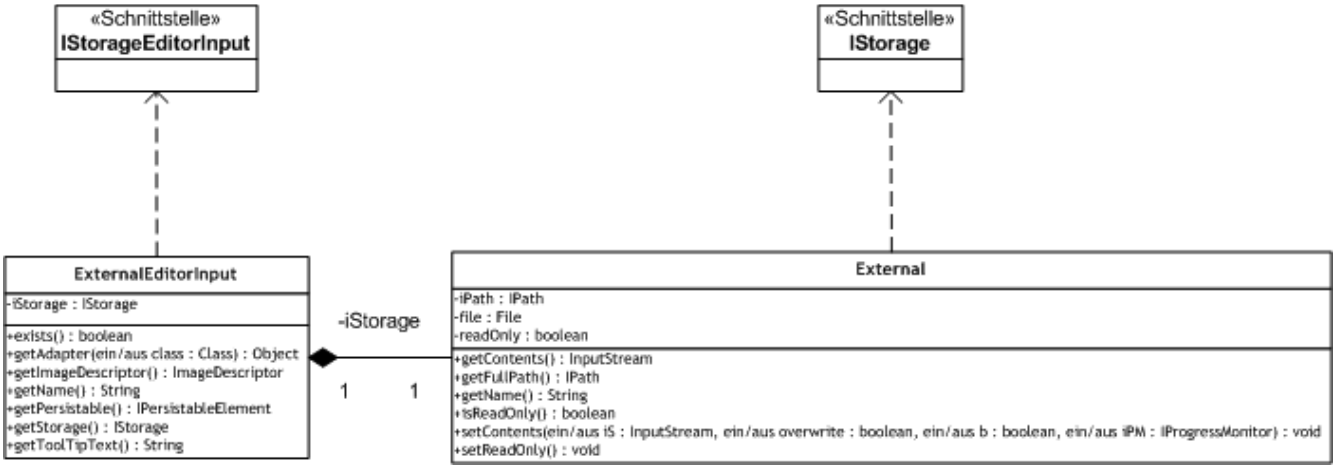

Abbildung 25: External- und ExternalEditorInput-Klassendiagramm

Nachdem die benötigten Klassen erstellt wurden muss der Dokument-Provider in Eclipse registriert werden. Hierzu wird in der Manifest-Datei der Erweiterungspunkt *org.eclipse.ui.editors.documentProviders* deklariert. Durch Rechtsklick auf den neuen Eintrag und Auswahl von NEW | PROVIDER wird der Provider hinzugefügt. In den Eigenschaften wird unter *class* der vollständige Klassenpfad zum Dokument-Provider *com.abaxx.workflow.eclipse.ExternalDocumentProvider* angegeben und unter *input Types* der unterstützte Eingabetyp *com.abaxx.workflow.eclipse.ExternalEditorInput*. Die *id* wird mit *com.abaxx.workflow.eclipse.externaldocumentprovider* belegt.

Dokumente werden mit Hilfe der Methode `openFile()` in der Klasse `EclipsePlugin` geöffnet. Diese Methode wird vom `ReceiveThread` angestoßen. Übergeben werden der Paketname inklusive Klassennamen, falls es sich um die Implementierung einer Aktivität handelt bzw. der vollständige Pfad zum Dokument, falls es sich um einen Part in einer XML-Datei handelt, und die Zeilennummer zu der gesprungen werden soll. Das Dokument wird zuerst mit Hilfe der `SearchEngine` des JDT gesucht. Falls es sich bei `filename` um einen Klassennamen handelt und diese Klasse in einem Projekt existiert findet die `SearchEngine` sie. War `filename` der vollständige Pfad zum Dokument so wird dieses nicht gefunden. Es wird dann die zweite Bedingung geprüft. Wenn eine Zeilennummer größer 0 übergeben wurde, handelt es sich bei dem Dokument um einen Part. Falls keine der Bedingungen zutrifft, handelt es sich um eine Datei, die nicht im Workspace von Eclipse vorkommt.

```
public void openFile(String filename, int lineNumber) throws CoreException {
...
	LocationSearchRequestor locationSearchRequestor = new LocationSearchRequestor();

	new SearchEngine().search(SearchPattern.createPattern(filename,
IJavaSearchConstants.TYPE, IJavaSearchConstants.DECLARATIONS,
SearchPattern.R_PATTERN_MATCH), new SearchParticipant[]
{ SearchEngine.getDefaultSearchParticipant() }, SearchEngine.createWorkspaceScope(),
locationSearchRequestor, null);

	if (locationSearchRequestor.getLocation() != null) {
			iPath = locationSearchRequestor.getLocation();

			iFile =
ResourcesPlugin.getWorkspace().getRoot().getFileForLocation(iPath);
	} else if (lineNumber > 0) {
			iPath = new Path(filename);

			iFile =
ResourcesPlugin.getWorkspace().getRoot().getFileForLocation(iPath);
	} else {
			iPath = new Path(filename);
	}
...
}
```

Listing 12: Suchen des Dokuments in den offenen Projekten

Nach dem Setzen des Pfades erfolgt das eigentliche Öffnen des Dokuments. Falls das Attribut `iFile` gesetzt ist, handelt es sich um ein Dokument in einem Eclipse-Projekt. Es können somit die von Eclipse zur Verfügung gestellten Dokument-Provider verwendet werden. Zuerst wird überprüft, welcher Editor dem zu öffnenden Dokumenttyp zugeordnet ist. Hierzu wird die EditorRegistry abgefragt. Daraufhin kann das Dokument geöffnet werden. Falls das Attribut `iFile` nicht gesetzt war, handelt es sich bei dem Dokument um ein externes Dokument. In diesem Fall werden die zuvor erstellten Klassen `External` und `ExternalEditorInput` zum Öffnen verwendet.

```
public void openFile(String filename, int lineNumber) throws CoreException {
...
      if (iFile != null && iFile.exists()) {
            IEditorDescriptor iEditorDescriptor =
this.getWorkbench().getEditorRegistry().getDefaultEditor(iFile.getName());

            String editorId;

            if (iEditorDescriptor == null) {
                  editorId = ''org.eclipse.ui.DefaultTextEditor'';
            } else {
                  editorId = iEditorDescriptor.getId();
            }

            iTextEditor = (ITextEditor) iWorkbenchPage.openEditor(new
FileEditorInput(iFile), editorId).getAdapter(ITextEditor.class);
      } else {
            External external = new External(iPath);

            iTextEditor = (ITextEditor) iWorkbenchPage.openEditor(new
ExternalEditorInput(external),
''org.eclipse.ui.DefaultTextEditor'').getAdapter(ITextEditor.class);
            ...
      }
...
}
```

Listing 13: Das eigentliche Öffnen des Dokuments

Falls es sich bei dem Dokument um eine Parts XML-Datei handelt, wird eine Zeilennummer größer 0 übergeben. In diesem Fall muss zu der entsprechenden Zeile im Editor gesprungen werden (siehe Listing 14).

```
	try {
		if (lineNumber > 0) {
			lineNumber--;

			IDocument iDocument =
iTextEditor.getDocumentProvider().getDocument(iTextEditor.getEditorInput());

			iTextEditor.selectAndReveal(iDocument.getLineOffset(lineNumber),
iDocument.getLineLength(lineNumber));
		}
	} catch (BadLocationException e) {
		DebugPlugin.log(e);
	}
}
```

Listing 14: Sprung zur gewünschten Zeilennummer

7.1.4 Implementierung einer Grundeinstellungsseite

In der Klasse `ReceiverThread` wird der Port, auf dem die lokale Socket-Verbindung geöffnet werden soll, aus dem `PreferenceStore` von Eclipse gelesen. Beim `PreferenseStore` handelt es sich um ein Objekt mit dessen Hilfe Grundeinstellungen innerhalb eines Plug-Ins gespeichert werden können. Die dort gespeicherten Werte werden von Eclipse verwaltet und sind auch über das Schließen von Eclipse hinaus persistent, so lange nicht der Arbeitsbereich (Workspace) geändert wird. Nach einer frischen Installation des Plug-Ins befindet sich natürlich noch kein Wert für den Port im `PreferenceStore`. Es muss deshalb ein PreferenceInitializer implementiert werden, der die Standardeinstellungen für bestimmte Werte beim Starten des Plug-Ins initialisiert. Hierzu wird ein neuer Erweiterungspunkt *org.eclipse.core.runtime.preferences* hinzugefügt. Diesem wird ein neuer *initializer* mit Angabe der Implementierungsklasse *abaxx.workflow.tools.eclipse.wfmlistener.core.runtime.ListenerPreferenceInitializer* hinzufügt. In dieser Klasse übernimmt die Methode `initializeDefaultPreferences()`

die eigentliche Arbeit und setzt die Standardeinstellungen (siehe Listing 15). Auf diese Weise können beliebig viele Einstellungen initialisiert werden. So lange der Benutzer diesen Wert in der Grundeinstellungsseite nicht verändert, wird der im PreferenceInitializer gesetzte Standardwert verwendet.

```
public void initializeDefaultPreferences() {
	IpreferenceStore preferenceStore =
WfmlistenerPlugin.getDefault().getPreferenceStore();

	preferenceStore.setDefault(''listeber_port'', ''7400'');
}
```

Listing 15: Initialisierung eines Standardwerts für eine Grundeinstellung

Damit der Benutzer die Porteinstellung verändern kann wird eine Grundeinstellungsseite implementiert. Hierzu wird die Erweiterung *org.eclipse.ui.preferencePages* hinzugefügt und dort das Element *page* angegegeben. Neben einer eindeutigen *id* wird der Grundeinstellungsseite ein *name* mit dem Wert *abaXX Process Modeler* und die Implementierungsklasse *abaxx.workflow.tools.eclipse.wfmlistener.ui.abaxxPreferencePage* hinzugefügt.

In der Implementierungsklasse wird innerhalb der Methode *createContents()* die Oberfläche der Grundeinstellungsseite aufgebaut. Der Wert für den Port wird aus dem PreferenceStore gelesen. In einer Grundeinstellungsseite hat der Benutzer die Möglichkeit auf *RestoreDefaults* und *Apply / OK* zu drücken. Im Falle von *RestoreDefaults* sollen wieder die Standarwerte für die Einstellungen gesetzt werden.

```
protected void performDefaults() {
	this.text.setText(this.getPreferenceStore().getDefaultString(''listener_port''));

	super.performDefaults();
}
```

Listing 16: Setzen der Standardwerte

Bei Drücken von *Apply / OK* sollen die eingegebenen Werte übernommen werden. Normalerweise schreibt Eclipse das PreferenceStore erst beim beenden der Anwendung wieder auf die Platte. Bei einem Absturz der Anwendung würde diese verloren gehen. Es empfiehlt sich deshalb die Speicherung der Werte selbst anzustoßen. Zuletzt muss noch der `ReceiverThread` neu gestartet werden, damit er auf dem neuen Port lauscht.

```
public boolean performOk() {
	this.getPreferenceStore().setValue(''listener_port'', this.text.getText());

	WfmlistenerPlugin.getDefault().savePluginPreferences();

	WfmlistenerPlugin.refreshReceiverThread();

	return true;
}
```

Listing 17: Speichern der Grundeinstellungswerte und Neustarten des ReceiverThreads

7.1.5 Installation des Plug-Ins

Um das Plug-In möglichst einfach installieren zu können, wird ein Feature erstellt, welches das Plug-In kapselt und über den Update Manager von Eclipse installiert werden kann. Mit Hilfe des Projekt-Assistenten wird ein neues Feature-Projekt angelegt. Als *Project name* wird *Wfmlistener* angegeben. Im nachfolgenden Schritt wird die *Feature ID* mit dem gleichen Wert belegt wie beim Plug-In: *abaxx.workflow.tools.eclipse.wfmlistener*. Der *Name* des Features ist *Process Modeler Listener* und der *Provider* wird mit *abaXX Technology AG* belegt. Im daraufhin erstellten Feature-Projekt befindet sich eine Datei mit dem Namen *feature.xml*. Durch Doppelklick kann diese in einem speziellen Editor geöffnet werden, ähnlich wie die *plugin.xml*. Über den Karteireiter *Plug-Ins* werden alle Plug-Ins angegeben, die Teil dieses Features sind. So lange diese auch Teil des Eclipse-Workspace sind, werden sie beim Drücken von *Add...* aufgeführt und können ausgewählt werden. Es empfiehlt sich auch die Abhängigkeiten des Features über den Karteireiter *Dependecies* anzugeben. Hierdurch überprüft Eclipse bei der Installation über den Update Manager ob alle benötigten Features und Plug-Ins installiert sind. Falls dies nicht der Fall ist erscheinen

Warnmeldungen auf die der Benutzer reagieren kann. In der feature.xml können desweiteren eine Beschreibung des Features und Copyright-Notizen sowie eine Lizenzvereinbarung definiert werden.

7.2 Implementierung des zweiten Schritts

Die Implementierung des zweiten Integrationsschritts bildete den Hauptteil der vorliegenden Diplomarbeit. Das Ziel war es, möglichst viele Komponenten des Process Modelers über die SWT-AWT-Brücke in Eclipse zu übernehmen. Es wurde nicht erwartet, dass am Ende der Diplomarbeit eine produktive Version fertig sein wird, da der Aufwand hierfür als zu groß eingeschätzt wurde.

Der Process Modeler besteht aus hunderten von Klassen in duzenden von Paketen. Zwischen diesen Klassen bestehen teils sehr komplexe Abhängigkeiten. Es ist deshalb nicht möglich alle Einzelheiten der Implementierung anzusprechen, geschweige denn jede einzelne Klasse. Im folgenden wird deshalb auf die besonders wichtigen und interessanten Punkte eingegangen. Die fertigen Plug-Ins und der Quellcode konnten wegen Lizenzfragen und Gründen der Diskretion nicht mit auf die beiliegende CD aufgenommen werden.

7.2.1 Architektur des neuen Produkts

Das erstellte Produkt besteht aus insgesamt vier Plug-Ins und einem Feature, das die Plug-Ins kapselt. Die Anwendung wurde in vier Plug-Ins aufgeteilt um eine bessere Strukturierung und Trennung des Quellcodes zu erreichen. Bei Eclipse Plug-Ins gibt es immer eine Art Abhängigkeitshierarchie, wobei keine Zyklen auftreten dürfen. Im Fall des Process Modeler bildet das Plug-In *abaxx.workflow.tools.eclipse.processmodeler.lib.3rdparty* die unterste Hierarchieebene. Dieses Plug-In dient als Container und kapselt Bibliotheken von Fremdherstellern, ant.jar, dom4j.jar oder velocity.jar sind nur einige Beispiele. Das

nächste Plug-In in der Hierarchie ist *abaxx.workflow.tools.eclipse.processmodeler.lib.abx*. Genauso wie das vorige dient es als Container, dieses mal für Bibliotheken von abaXX. Das dritte Plug-In ist *abaxx.workflow.tools.eclipse.processmodeler*, es stellt den Kern des Process Modelers dar. Alle Klassen ohne irgendwelche Abhängigkeiten zur Benutzeroberfläche kommen in dieses Plug-In, so zum Beispiel das Process Modeler Nature. Das letzte Plug-In heißt *abaxx.workflow.tools.eclipse.processmodeler.ui* und enthält alle übrigen Klassen. Zur Zeit befinden sich die meisten Klassen in diesem Plug-In, da sie vielfach Abhängigkeiten untereinander haben und eine Auslagerung in das Kern Plug-In zu Zyklen führen würde. In Zukunft sollen diese Abhängigkeiten reduziert und die Klassen besser aufgeteilt werden.

Abbildung 26: Plug-In Hierarchie und Abhängigkeiten

Durch die Kapselung der Bibliotheken in Plug-Ins wird die Abhängigkeit zum Process Modeler Verzeichnis beseitigt. Das Produkt kann in jede Eclipse Installation an beliebiger Stelle auf der Festplatte installiert werden. Einige Abhängigkeiten, zum Beispiel zur Lizenzdatei oder Konfigurationsdatei, konnten nicht vollständig aus dem Quellcode entfernt werden. Deshalb wird während der Installation der Plug-Ins ein Install-Handler gestartet, der die Pfade zu diesen Dateien vom Benutzer abfragt. In Zukunft soll die Eclipse Integration vollständig unabhängig vom derzeitigen Process Modeler funktionieren. Die Möglichkeit der Integration eines Install-Handlers ist ein relativ selten benutztes Feature in Eclipse. Die Einbindung erwies sich als nicht einfach, vor allem wegen unzureichender Dokumentation.

Ein mit dem Plug-In Development Environment (PDE) erstelltes Feature inklusive Plug-Ins lässt sich relativ einfach aus Eclipse heraus in eine installierbare Form exportieren. Dieser Vorgang wird manuell vorgenommen. Bei der abaXX Technology AG wird täglich ein automatischer Build aller Produkte aus dem CVS durchgeführt. Auch die Process Modeler Eclipse Integration sollte Teil dieses automatischen Builds werden, so dass der Export-Vorgang automatisiert werden musste. Dies geschah mit Hilfe von Ant-Skripten. Jedes Plug-In verfügt über ein eigenes Ant-Skript, das alle Dateien aus dem CVS herausholt, bei Bedarf eine Kompilierung der Klassen realisiert und zum Schluss den Export in eine installierbare Form durchführt. Diese einzelnen Skripte werden durch das Ant-Skript des Features angestoßen und gesteuert.

7.2.2 Benutzeroberfläche

Die Benutzeroberfläche wird durch die Process Modeler Perspektive vorgegeben. Sie dient der Aufteilung des zur Verfügung stehenden Platzes und der Positionierung der benötigten Ansichten an den gewünschten Stellen. Bei der Anordnung der einzelnen Komponenten wurde versucht das bereits vom Process Modeler bekannte Layout weitestgehend beizubehalten, so dass sich der Benutzer nicht völlig neu orientieren muss. Zur Navigation durch die Ressourcen stehen auf der linken Seite der Navigator der Eclipse Plattform und der Package Explorer des JDT zur Verfügung. Links darunter wurde die Outline-Ansicht eingefügt. Zur Zeit wird sie von den Komponenten des Process Modelers noch nicht verwendet, hilft aber beim Editieren von Java-Quellcode aus dem die Implementierungen der einzelnen Aktionen bestehen. In der Mitte, wie von Eclipse bekannt, werden die Editoren geöffnet, so auch der Process Modeler Editor zur Modellierung der Geschäftsprozesse. Auf der rechten Seite wurden die Gallerien des Process Modelers untergebracht: Activity Gallery, Process Gallery, Parts Gallery, Object Gallery und Participant Gallery. Darunter befindet sich die Overview in der eine verkleinerte Darstellung des modellierten Prozesses dargestellt wird. Vergleichbar einer Map. In der Mitte, unter dem Bereich wo die Editoren eingeblendet werden, stehen der Property Editor und die Messages-Ansicht zur Verfügung.

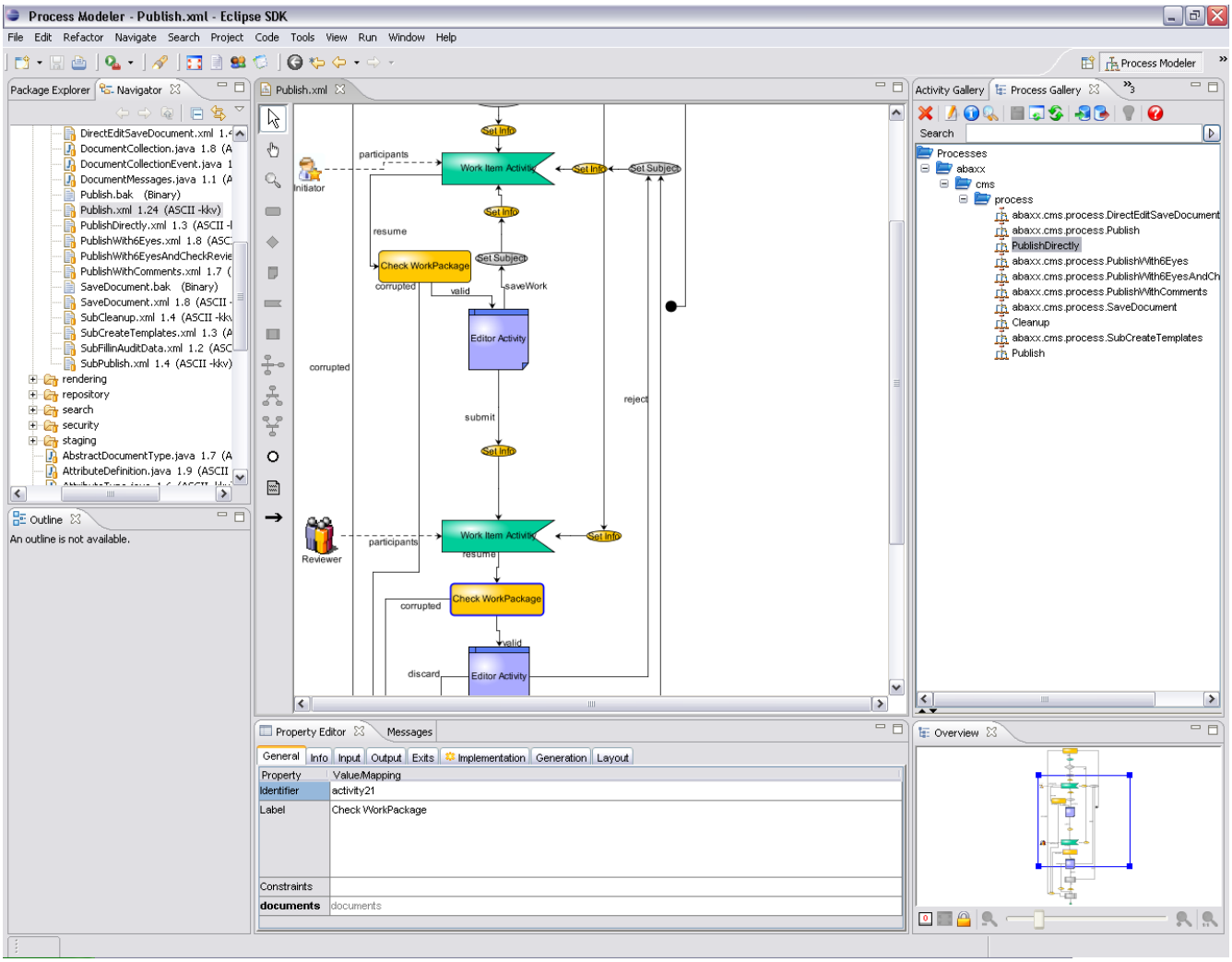

Abbildung 27: Process Modeler Perspektive

7.2.3 Editor

Die zentrale Komponente des Process Modelers ist der Editor zum Modellieren der Prozesse. Er baut zum großen Teil auf JViews-Komponenten auf und ist eng mit diesen verknüpft. Nach dem Öffnen eines Prozesses befindet sich dessen Inhalt (das Modell) im Editor. Die Basis des Editors in der Swing-Anwendung bildet die Klasse `WfmFrame`. Es handelt sich hierbei nicht um einen `JFrame` oder `Frame`, wie der Name suggeriert. `WfmFrame` enthält das Modell des Prozesses, das durch die Klasse `WdaModel` abgebildet wird. Jeglicher Zugriff auf einen offenen Prozess und dessen Elemente erfolgt über den gerade aktiven Editor.

Im frühen Stadium der Integration kristallisierten sich zwei Möglichkeiten heraus, wie der Editor in Eclipse übernommen werden könnte. Bei der ersten gäbe es einen Eclipse-Editor mit einem Attribut `WfmFrame` in dem der komplette Editor der Swing-Anwendung ohne größere Änderungen hätte gespeichert werden können. Alle Zugriffe auf den offenen Prozess und dessen Elemente würden vom Eclipse-Editor einfach nur weitergereicht an die Klasse `WfmFrame`. Die zweite Möglichkeit bestand darin, die komplette Funktionalität von `WfmFrame` in einen Eclipse-Editor zu übernehmen. Dies bedeutete natürlich viel mehr Aufwand als bei der ersten Möglichkeit, ergäbe aber eine schönere Lösung. Nach reiflicher Überlegung wurde die zweite Möglichkeit verwirklicht. Hauptsächlich um einen soliden Grundstein für die weitere Entwicklung zu legen.

Ein Problem, das während der Integration in Eclipse auftrat, war der Unterschied im Zeitpunkt der Aktivierung des Editors zwischen der Swing-Anwendung und Eclipse. Bei der Swing-Anwendung wird zuerst der `WfmFrame` erzeugt und aktiv gesetzt. Anschließend wird das Modell erstellt und gleich danach initialisiert. Es gibt nun Klassen, die während der Initialisierung des Modells bereits auf den aktiven Editor (genauer gesagt auf das teilweise initialisierte Modell) zugreifen. In Eclipse hingegen wird der Editor erst aktiv gesetzt, nachdem das Modell komplett initialisiert ist. Die Zugriffe auf den gerade aktiven Editor während der Initialisierung des Modells laufen also entweder ins leere oder greifen auf einen falschen Editor zu, da zu diesem Zeitpunkt ein anderer Editor gerade aktiv sein kann. Um diesem Problem entgegenzuwirken wurde ein Workaround entwickelt. Nach der Erstellung des Modells wird dieses in der Klasse `ProcessModelerUi` gespeichert, der zentralen Klasse der Process Modeler Eclipse Integration. Die Klassen, die während der Initialisierung des Modells auf den aktiven Editor zugreifen, wurden so umgeschrieben, dass sie nicht mehr auf das Modell im aktiven Editor, sondern auf das in der Klasse `ProcessModelerUi` gespeicherte Modell zugreifen. Auf diese Weise bekommen sie Zugriff auf das Modell, welches sich gerade in der Initialisierung befindet. Dieses Problem sollte in Zukunft sauberer gelöst werden. Hierzu müssen aber größere Eingriffe in einige Klassen vorgenommen werden.

7.2.4 Ansichten

Die Galerien des Process Modelers wurden in einzelne Eclipse-Ansichten eingebettet. So lange kein Process Modeler Editor offen ist, bleiben die Galerien leer und zeigen nur eine Nachricht an, dass die Galerie im Augenblick nicht verfügbar ist. Erst nach dem Öffnen eine Process Modeler Editor werden Inhalte in den Galerien angezeigt. Bei einem Wechsel zu einem anderen Editor, zum Beispiel dem Java Editor, werden die Galerien wieder deaktiviert.

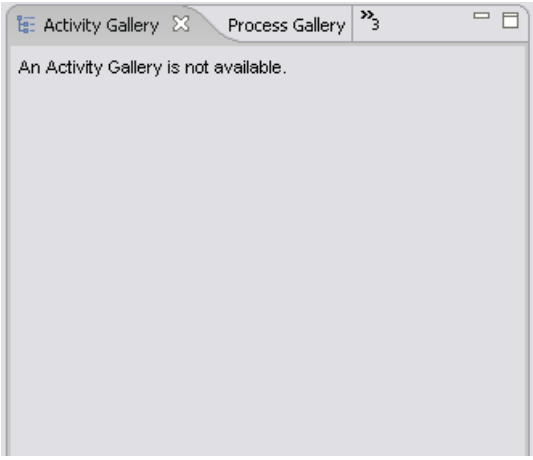

Abbildung 28: Galerie nicht verfügbar

Die Projektstruktur der eigenständigen Swing-Anwendung unterscheidet sich stark von der Eclipse Projektstruktur. In der Swing-Anwendung gibt es immer nur ein aktives Projekt, so dass die Galerien immer diesem Projekt zugeordnet sind. Um ein anderes Projekt zu öffnen muss das alte geschlossen werden. In diesem Fall werden die Galerien für das neue Projekt geladen. Bei Eclipse können alle offenen Projekte als aktive Projekte angesehen werden. Der Wechsel zwischen zwei Projekten geschieht schon beim Öffnen einer Datei aus einem anderen Projekt. Die Galerien mussten diesen Umstand berücksichtigen. Wenn ein Prozess aus einem anderen Projekt geöffnet wird, müssen sie die Inhalte des neuen Projekts widerspiegeln. Hierzu wurde die Klasse `RefreshProjectAction` erstellt, die im Hintergrund bei bestimmten Tätigkeiten ausgeführt wird und überprüft ob sich das Projekt geändert hat. Falls dies der Fall ist, werden die Galerien aktualisiert. Damit diese nicht jedes mal neu aus den XML-Dateien gelesen werden müssen, was zu großen Performance-

Einbussen bei öfteren Projektwechseln führen würde, wurde ein Caching-Mechanismus erarbeitet. Beim ersten Öffnen eines Projekts werden dessen Galerien von der Platte gelesen. Bei jeder folgenden Aktivierung des Projekts werden die Inhalte nur noch aus dem Cache geladen. Dieser Mechanismus wurde auch für die Projektkonfiguration implementiert. Der Process Modeler verfügt über die Klasse `Configuration`, werlche die aktuelle Konfiguration des aktiven Projekts widerspiegelt. Auch diese Konfiguration muss bei jedem Projektwechsel aktualisiert werden.

Die Galerien im Process Modeler können editiert werden. Nach dem Editieren müssen sie gespeichert werden. Dies widerspricht dem Konzept von Eclipse, nach dem die Änderungen in Ansichten sofort wirksam werden sollten. Probleme können dann entstehen, wenn Änderungen in einer Galerie gemacht und nicht gespeichert werden und anschließend zu einem anderen Projekt gesprungen wird. Wenn Eclipse nun geschlossen wird, gehen ungespeicherte Änderungen verloren. Um dies zu vermeiden, wurde ein Mechanismus eingeführt der beim Schliessen von Eclipse alle Galerien auf ungespeicherte Änderungen hin überprüft, ähnlich der Funktionalität von Editoren. Dies soll aber nicht immer so bleiben. Es wird bereits nach einer anderen Lösung gesucht. Eine Möglichkeit ist die Änderungen sofort zu übernehmen. Eine andere die Implementierung eines zusätzlichen Editors für die XML-Dateien der Galerien. Die Ansichten selbst würden dann nur noch Inhalte darstellen, ohne dass diese dort editierbar wären.

7.2.5 Aktionen

Die Funktionen der Swing-Anwendung wurden gleichmäßig, der Zugehörigkeit entsprechend, auf die einzelnen Menüs von Eclipse verteilt. Zusätzlich wurden vier neue Menüs implementiert: Refactor, Code, View und Tools. Die meisten Aktionen werden erst dann aktiviert, wenn ein Process Modeler Editor offen ist. Andernfalls werden sie erst gar nicht angezeigt.

Viele Aktionen in Eclipse sind so genannte retargetable Actions. Dies bedeutet, dass ein Eintrag im Menü von mehreren Plug-Ins verwendet werden kann. Ein konkretes Beispiel wäre zum Beispiel die Aktion *Copy* im *Edit*-Menü. Wenn im Navigator eine Datei ausgewählt wurde, kann diese kopiert und anschließend eingefügt werden. Wenn aber zum Beispiel gerade im Java-Editor gearbeitet wird und dort eine Passage ausgewählt wird, kann diese über den gleichen Menüeintrag kopiert werden. Dies funktioniert, da in diesem Augenblick eine andere Klasse für die Abarbeitung der Aktion zuständig ist. Die Klassen werden von Eclipse je nach aktiviertem Editor oder Ansicht gewechselt. Hierzu muss über eine spezielle Erweiterung ein so genannter `EditorActionBarContributor` eingebunden werden.

7.2.6 Erstellung eines neuen Projekts und neuer Prozesse

Für die Erstellung eines neuen Process Modeler Projects und neuer Prozesse wurden Assistenten erstellt, die sich nahtlos in Eclipse an den entsprechenden Stellen integrieren. Zusätzlich wurde die Möglichkeit geschaffen, bereits existierende Projekte zu Process Modeler Projects konvertieren zu können. Hierzu wurde ein zusätzlicher Eintrag in das Kontextmenü eines Projekts hinzugefügt. Dieser ist nur dann sichtbar, wenn es sich nicht um ein Process Modeler Project handelt. Eine ähnliche Funktionalität bietet das PDE an, das in diesem Fall als Vorbild genommen wurde.

Die Erstellung eines neuen Prozesses funktioniert in Eclipse anders als in der Swing-Anwendung. Dort wurde ein neuer Editor geöffnet und erst wenn der Anwender diesen speicherte wurde der Prozess als Datei auf die Festplatte geschrieben. Bis dahin existierte der Prozess nur als Objekt in der JVM. In Eclipse wird beim Erstellen eines neuen Prozesses zuerst die Datei mit dem im Assistenten angegebenen Namen auf der Festplatte erstellt. Im Anschluss wird diese Datei in einem Process Modeler Editor geöffnet. Diese Änderung war notwendig, da einem Editor in Eclipse immer eine Datei (Ressource) zugeordnet werden muss. Wegen dieser Architekturänderung mussten einige andere Klassen, die vom vorherigen Verhalten abhingen, angepasst werden.

7.2.7 Einstellungen und Eigenschaften

Die Einstellungs- und Eigenschaftsseiten des Process Modelers wurden entsprechend ihrer Zugehörigkeit in Eclipse verteilt. Die Einstellungen für die gesamte Anwendung wurden unter WINDOW -> PREFERENCES... implementiert. Die Eigenschaften eines Projekts sind über das Kontextmenü unter dem Menüpunkt *Properties* einsehbar. Gleiches gilt für die Eigenschaften eines Prozesses. Über spezielle Filter bei der Definition der Erweiterung lässt sich bestimmen, wann genau eine Eigenschaftsseite angezeigt werden soll. Denn sie soll nicht erscheinen, wenn die *Properties* einer XML-Datei aufgerufen werden.

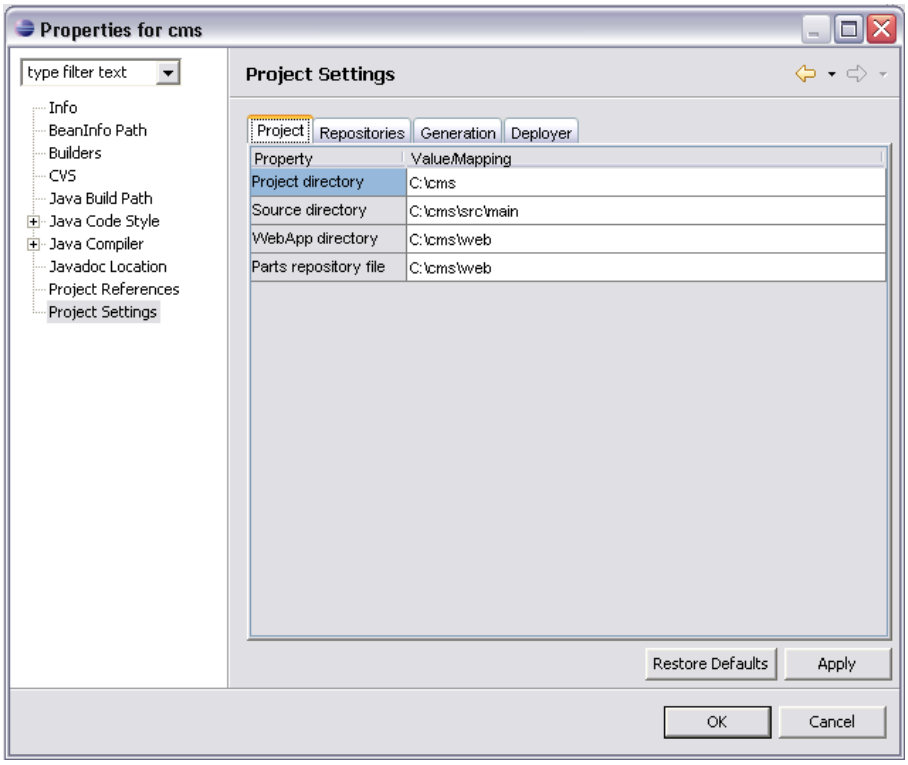

Abbildung 29: Projekteinstellungen des Process Modelers

Bei allen Einstellungs- und Eigenschaftsseiten wurden die kompletten Swing-Komponenten mit ihrer Tab-Struktur übernommen (siehe Abbildung 29). Dies erscheint einem zwar fremd im Eclipse-Layout, eine Umprogrammierung hätte aber zu viel Zeit in Anspruch genommen. Für die Zukunft ist es angedacht, die einzelnen Tabs auf getrennte Unterseiten des Eclipse-Eigenschaftsdialogs aufzuteilen.

7.2.8 Threading-Probleme

Während der gesamten Implementierungsphase musste mit Synchronisationsproblemen zwischen dem AWT- und dem SWT-Thread gekämpft werden. Denn durch die Verwendung zweier unterschiedlicher Toolkits für die Darstellung der grafischen Objekte laufen im Hintergrund zwei separate Rendering-Threads. Es wurden verschiedene Lösungsmöglichkeiten ausprobiert um das Problem zu lösen. Am Ende wurde eine Lösung für den Zugriff auf den SWT-Thread gefunden, die das Problem zufriedenstellend löst. Der Zugriff auf den AWT-Thread gestaltet sich leider immer noch schwierig und muss von Fall zu Fall gelöst werden.

Der Zugriff auf SWT-Komponenten erfolgt so gut wie immer über von Eclipse bereitgestellte Methoden. Zum Beispiel auf den gerade aktiven Editor oder eine bestimmte Ansicht. In einigen Fällen wird auch von einem bereits offenen Fenster eine `Shell` benötigt, um Dialoge modal anzeigen zu lassen. Der direkte Zugriff auf die SWT-Komponenten über die Eclipse-Methoden endet in einer `InvalidThreadAccess`-Exception, wenn der Zugriff aus dem AWT-Thread erfolgt. Um dies zu verhindern, gibt es die Methode `Display.asyncExec(Runnable runnable)`. Dies entspricht der Methode `SwingUtilities.invokeLater()` in Swing. Mit Hilfe dieser Methode wird ein Aufruf von Code durch den SWT-Thread erzwungen, wobei das Hauptprogramm weiterläuft. Hierbei gibt es nun zwei Probleme. Zuerst einmal muss jeder Aufruf entsprechend verpackt werden, was auf Dauer sehr ermüdend sein kann. Zweitens ist nicht immer sicher, aus welchem Thread der Zugriff gerade erfolgt. Denn ein Aufruf von `Display.asyncExec(Runnable runnable)` aus dem SWT-Thread bringt auch Exceptions mit sich. Die von Eclipse bereitgestellten Methoden wurden deshalb in der Klasse `ProcessModelerUi` adaptiert und um zusätzliche Abfragen ergänzt, die den aktuellen Thread überprüfen und dementsprechend reagieren. Im Folgenden ein Beispiel für den Zugriff auf den gerade aktiven Editor (siehe Listing 18). Vergleichbare Methoden wurden für den Zugriff auf Ansichten implementiert.

```
public static ProcessModelerEditor getActiveProcessModelerEditor() {
	if (Thread.currentThread().getName().startsWith(''AWT'')) {
		Display.getDefault().asyncExec(new Runnable() {
			public void run() {
				IEditorPart activeEditor =
PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().getActiveEditor();

				if (activeEditor instanceof ProcessModelerEditor) {
					processModelerEditor = (ProcessModelerEditor) activeEditor;
				} else {
					processModelerEditor = null;
				}
			}
		});
	} else {
		IEditorPart activeEditor =
PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().getActiveEditor();

		if (activeEditor instanceof ProcessModelerEditor) {
			processModelerEditor = (ProcessModelerEditor) activeEditor;
		} else {
			processModelerEditor = null;
		}
	}

	return processModelerEditor;
}
```

Listing 18: Adaptierter Zugriff auf den gerade aktiven Editor

Zusammenfassung

„Vieles hätte ich verstanden, wenn man es mir nicht erklärt hätte."

- Stanislaw Jerzy Lec -

Eclipse erwies sich als durchdachtes und performantes Framework mit viel Potential für die Zukunft. Vor allem die modulare Plug-In-Architektur ist sehr ausgeklügelt und ermöglicht eine nahtlose Integration neuer Komponenten in das Gesamtsystem. Wie gut eine Integration letztendlich ist, hängt in großem Maße vom investierten Zeitaufwand und der Erfahrung des Entwicklers ab. Denn Vielzahl der Erweiterungsmöglichkeiten lässt sich nur nach und nach erfassen. Das Hinzufügen neuer Funktionalitäten über Erweiterungen reichte von sehr einfach bis komplex und schwierig. Der Grund hierfür lag aber oft in zu oberflächlicher Dokumentation. Vor allem Themenbereicht, die nicht oft verwendet werden, sind teilweise extrem schlecht dokumentiert.

Die Integration von Swing-Anwendungen über die SWT-AWT-Brücke im speziellen, erwies sich als gute Möglichkeit für einen ersten Schritt bei der Einbindung der eigenen Anwendung in Eclipse. Sie erspart eine Menge Zeit und ist von der Qualität mehr als nur ausreichend. Das einzig größere Problem ist die Synchronisation der beiden Rendering-Threads von AWT und SWT, das aber im großen und ganzen lösbar ist. Eine gute Erweiterung der SWT-AWT-Brücke von Eclipse wären Klassen, die bestimmte Zugriffe adaptieren und in Abhängigkeit vom aktuellen Thread ausführen, ähnlich der Lösung im zweiten Integrationsschritt.

Die anvisierten Ziele vom Anfang wurden im Rahmen der Diplomarbeit alle erreicht. Es wurde ein Integrationsszenario mit schrittweiser Überführung des Process Modelers in Eclipse erstellt. Die ersten beiden Schritte wurden bereits während der Diplomarbeit in Angriff genommen. Schritt eins konnte vollständig realisiert werden. Schritt zwei wurde zu etwa 80% fertig gestellt. Das in der Einleitung aufgeworfene Ziel der Erstellung eines

Prototypen wurde somit mehr als erfüllt. Zum Ende der Diplomarbeit existiert bereits eine Alpha-Version, die Bestandteil des nächsten Produkt-Releases wird. Schon jetzt erfüllen einige Erweiterungen den Wunsch nach mehr Produktivität. Das Öffnen der Implementierung von Aktivitäten und Parts aus der Mollierung heraus kam beispielsweise sehr gut bei den Mitarbeitern im Unternehmen an. Viele hatten sich diese Funktionalität schon seit langem gewünscht. Grundlage der zukünftigen Entwicklung bilden die Schritte drei bis fünf des aufgestellten Integrationsszenarios.

In der Zeit als die Diplomarbeit entstand, wurde ein neues Eclipse-Projekt vorgeschlagen: die Java Workflow Toolbox. Zur Zeit befindet sich das Projekt in einer frühen Phase, in der es versucht eine Gemeinde aufzubauen. Ziele der JWT sind der Aufbau eines Workflow Editors (WE) und eines Workflow engine Administration and Monitoring tools (WAM). Dies sind Gebiete, in denen sich auch der Process Modeler bewegt. Vor allem der geplante Workflow Editor könnte für den Process Modeler in Zukunft von Interesse sein. Es ist geplant den Workflow Editor auf Basis des GEF zu erstellen. Als Notation für die grafische Darstellung der Prozesse soll die Business Process Modeling Notation (BPMN) verwendet werden, als Beschreibungssprache die XML Process Definition Language (XPDL). Eine Verknüpfung des Process Modelers mit der JWT könnte in Zukunft einige Vorteile mit sich bringen und die Weiterentwicklung der Anwendung beschleunigen.

Glossar

Ansicht (View) Eine Ansicht ergänzt einen Editor in Eclipse und bietet zusätzliche Informationen an.

API Das Application Programming Interface (API) ist eine Programmierschnittstelle die vom Betriebssystem oder anderen Softwaresystemen weiteren Programmen zur Verfügung gestellt wird. Auf diese Weise werden verschiedene Funktionen zur weiteren Verwendung freigegeben.

AWT Das Abstract Window Toolkit (AWT) stellt eine Standard-API zur Erzeugung und Darstellung einer plattformunabhängigen grafischen Benutzerschnittstelle (GUI) für Java-Programme dar. Es zählt zu den schwergewichtigen (heavyweight) Frameworks zur Darstellung von Steuerelementen, da es die nativen GUI-Komponenten des Betriebssystems verwendet.

Design Pattern Als Design Pattern wird ein Entwurfsmuster bezeichnet, das eine erfolgreiche, generische Lösung für ein Problem in der Softwareentwicklung bietet. Es stellt somit eine wiederverwendbare Vorlage für bestimmte Probleme dar.

Draw2d Draw2d bietet einen Baukasten zum Ausrichten und Rendern von grafischen Objekten auf Basis von SWT.

EPL Die Eclipse Public License (EPL) ist die Lizenz unter der Eclipse vertrieben wird. Bei ihr handelt es sich um eine zur Open Source

Initiative (OSI) kompatible Lizenz, die eine effiziente kommerzielle Nutzung erlaubt.

Framework Ein Framework gibt in der Regel eine Anwendungsarchitektur vor. Der Entwickler registriert konkrete Implementierungen, die durch das Framework gesteuert und benutzt werden, statt lediglich Klassen und Funktionen zu benutzen. Der Begriff taucht insbesondere im Rahmen der objektorientierten und komponentenbasierten Entwicklung auf.

GEF Das Graphical Editing Framework (GEF) erlaubt die Entwicklung von grafischen Editoren auf Basis von SWT. Zum Ausrichten und Rendern der Grafiken wird Draw2d verwendet. Das GEF setzt eine zusätzliche Schicht zum Manipulieren (Skalieren, Verschieben usw.) der grafischen Objekte drauf.

GUI Unter dem Graphical User Interface (GUI) wird die Benutzeroberfläche einer grafischen Anwendung verstanden.

IBM International Business Machines (IBM) ist eines der ältesten IT-Unternehmen und Hauptinitiator von Eclipse.

IDE Das Integrated Development Environment (IDE) ist eine integrierte Entwicklungsumgebung zur Erstellung von Software. Sie bieten in den meisten Fällen Texteditoren, Compiler bzw. Interpreter, Linker, Debugger und Quelltextformatierungsfunktionen.

JAR Bei einem Java Archive (JAR) handelt es sich um ein Archivformat mit dessen Hilfe einzelne Klassen zu einem Paket gepackt werden. Es handelt sich dabei um eine ZIP-Datei mit zusätzlichen Metadaten.

JRE Die Java-Laufzeitumgebung besteht aus der JVM und den Java Klassenbibliotheken, welche die Standard-Klassen von Java zur Verfügung stellen. Sie werden miteinander abgestimmt und sind für verschiedene Betriebssysteme erhältlich.

JViews JViews von ILOG ist ein Framework zur Erstellung grafischer Oberfläche wie Editoren oder Karten, das auf Swing aufbaut.

JVM Die Java Virtual Machine (JVM) ist innerhalb des Java Runtime Environment (JRE) die für die Ausführung des Java-Bytecodes verantwortliche Komponente. Sie dient dabei als Schnittstelle zum Betriebssystem. Die JVM ist für verschiedene Betriebssysteme verfügbar. Erst durch sie wird der kompilierte Java-Code unabhängig vom Betriebssystem.

MVC Der Model-View-Controller ist ein Entwurfsmuster das zu den Architekturmustern zählt. Es trennt eine Anwendung in die drei Bereiche Datenmodell (Model), Präsentation (View) und Programmsteuerung (Controller). Hierdurch wird das Design der Applikation flexibel gehalten und die Wiederverwendbarkeit der einzelnen Komponenten erhöht.

OSGi Das OSGi-Framework ist eine offene, modulare und skalierbare Service Delivery Plattform auf Java-Basis. Sie ermöglicht als Basisplattform die nachträgliche Auslieferung und Installation von Diensten zur Laufzeit. Die Plug-Ins von Eclipse sind seit Version 3 OSGi Bundles.

OSI Die Open Source Initiative (OSI) ist eine Organisation, die sich der Förderung von Open-Source-Software widmet.

Perspektive (Perspective) Mit Hilfe einer Perspektive lässt sich in Eclipse die Benutzeroberfläche in einer bestimmten Art und Weise konfigurieren. So kann beispielsweise definiert werden welche Ansichten angezeigt werden sollen und an welcher Stelle. Es kann auch beeinflusst werden ob bestimmte Aktionen angezeigt werden sollen oder nicht.

plugin.xml In dieser Meta-Datei werden unter Eclipse alle Erweiterungen, die ein Plug-In zur Verfügung stellt, definiert. Hierdurch wird eine Trennung von Definition und Implementierung erreicht und die Funktionalität kann bei Bedarf zur Laufzeit geladen werden (Lazy-Loading), was Performance Vorteile mit sich bringt.

RCP Die Rich Client Plattform (RCP) stellt im Vergleich zur normalen Eclipse Plattform nur eine Teilmenge der Plug-Ins zur Verfügung und ist nicht in Richtung IDE orientiert. Sie stellt eine Basis-Plattform für die Entwicklung verschiedenster Anwendungen bereit.

Swing Swing ist die Nachfolge-Bibliothek von AWT zur Erstellung von grafischen Benutzeroberflächen. Im Unterschied zu AWT implementiert sie alle Komponenten in Java und ist unabhängig von den nativen Komponenten des Betriebssystems.

SWT SWT ist die von IBM entwickelte Alternative zu AWT und Swing. Sie geht einen ähnlichen Weg wie AWT und nutzt die nativen Komponenten des Betriebssystems. Da sich die Implementierung auf Java-Ebene aber von Plattform zu Plattform unterscheidet ist SWT nicht plattformunabhängig.

XML Die Extensible Markup Language (XML) ist ein Standard zur Erstellung maschinen- und menschenlesbarer Dokumente in Form einer Baumstruktur.

Abbildungsverzeichnis

Literaturverzeichnis

abaXX1, 2005 abaXX (Hg.): The abaXX.workflow Engine. Introduction & Overview. 2005. Stand: 01.02.2006

abaXX2, 2004 abaXX (Hg.): abaXX.components 4. 2004. URL. http://www.abaxx.de/abaxx/-?$part=German.Products. Stand: 01.02.2006

abaXX3, 2004 abaXX (Hg.): abaXX process.component. 2004. URL: http://www.abaxx.de/abaxx/-?$part=German.Products.common.ContentDetail&externalid=cnt:2668. Stand: 01.02.2006

abaXX4, 2005 abaXX (Hg.): Concepts & Terminology. Essentials of abaXX.workflow. 2005. Stand: 01.02.2006

Amsden, 2001 Amsden, Jim: Levels Of Integration. Five ways you can integrate with the Eclipse Platform. 03/2001. URL: http://www.eclipse.org/articles/Article-Levels-Of-Integration/Levels%20Of%20Integration.html. Stand: 26.10.2005

Bokowski, 2005 Bokowski, Boris: GEF. Maßgeschneiderte grafische Editoren mit GEF. Präsentation. Heidelberg / Deutschland. 06/2005

Edgar, 2004 Edgar, Nick / Haaland, Kevin / Li, Jin / Peter, Kimberley: Eclipse User Interface Guidelines Version 2.1. 02/2004. URL: http://www.eclipse.org/articles/Article-UI-Guidelines/Index.html. Stand: 24.02.2006

GEF, 2005 IBM (Hg.): GEF and Draw2d Plug-in Developer Guide. 02/2005. URL: http://help.eclipse.org/help31/topic/org.eclipse.gef.doc.isv/guide/guide.html. Stand: 30.01.2006

ILOG, 2002 ILOG (Hg.): ILOG JViews 5.5. Graphics Framework User's Manual. Frankreich. 12/2002

Majewski, 2004 Majewski, Bo: A Shape Diagram Editor. 08/2004. URL: http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html. Stand: 17.10.2005

Rutishauser, 2003 Rutishauser, Simon: Standard Widget Toolkit. Einführung in SWT anhand der Erstellung eines Brennprogramms. Maturarbeit. Bern-Kirchenfeld. 2003

Schill, 2005 Schill, Philipp: Eclipse as client container for J2EE applications. Diplomarbeit. Stuttgart / Deutschland. 02/2005

Shavor, 2004 Shavor, Sherry / D'Anjou, Jim / Fairbrother, Scott / Kehn, Dan / Kellerman, John / McCarthy, Pat: Eclipse. Anwendungen und Plug-Ins mit Java entwickeln. München / Deutschland. 2004

Spektrum, 2005 SIGS-DATACOM GmbH (Hg.): Eclipse wird zum Renner. Deutschland. 2005. In: Java Spektrum. Sonderheft Eclipse 2005

Stal, 2005 Stal, Michael: Eclipse 3.1. Interview mit Erich Gamma. Deutschland. 2005. In: Java Spektrum. Sonderheft Eclipse 2005

Ullenboom1, 2006 Ullenboom, Christian: Grafikprogrammierung mit dem AWT. URL: http://www.galileocomputing.de/openbook/javainsel5/javainsel14_000.htm. Stand: 14.02.2006

Ullenboom2, 2006 Ullenboom, Christian: Fenster unter grafischen Oberflächen. URL: http://www.galileocomputing.de/openbook/javainsel5/javainsel14_002.htm. Stand: 14.02.2006

Ullenboom3, 2006 Ullenboom, Christian: Komponenten im AWT und in Swing. URL: http://www.galileocomputing.de/openbook/javainsel5/javainsel15_002.htm. Stand: 14.02.2006